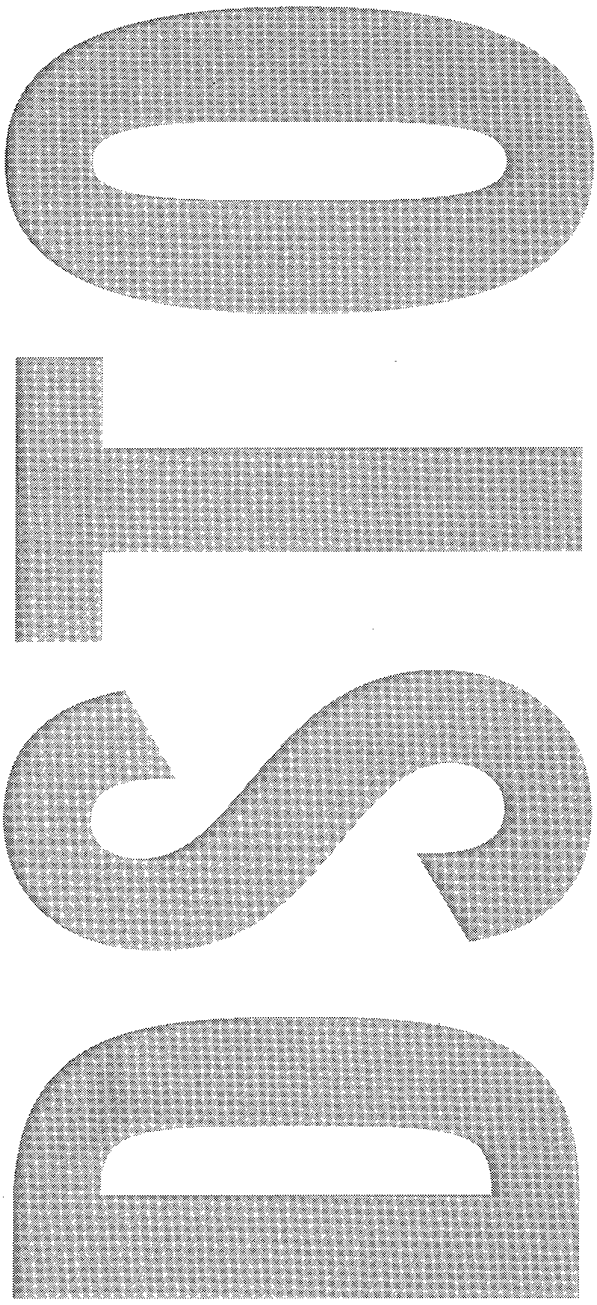
DEPARTMENT OF DEFENCE
DEFENCE SCIENCE & TECHNOLOGY ORGANISATION
DSTO



A Primer of CORBA: A Framework for Distributed Applications in Defence

T.A. Au

DSTO-GD-0192

A Primer of CORBA: A Framework for Distributed Applications in Defence

T. A. Au

Communications Division
Electronics and Surveillance Research Laboratory

DSTO-GD-0192

ABSTRACT

Based on object technology, the OMG defines an Object Management Architecture (OMA) for the support of interoperable applications across heterogeneous computing platforms. The communication core of this underlying model is the Common Object Request Broker Architecture (CORBA) that provides a framework for flexible and transparent communication between distributed objects. The adoption of this approach eases software development by allowing interaction between reusable components through well-defined interfaces. In particular, applying CORBA technology to C4I problems in the military environment provides simple integration of legacy software and COTS software. This report provides an overview of the OMA, and describes in detail each component of CORBA.

DEPARTMENT OF DEFENCE
DEFENCE SCIENCE & TECHNOLOGY ORGANISATION
DSTO

Published by

DSTO Electronics and Surveillance Research Laboratory
PO Box 1500
Salisbury South Australia 5108 Australia

Telephone: (08) 8259 5555
Fax: (08) 8259 6567

AR-010-622
March 1999

APPROVED FOR PUBLIC RELEASE

A Primer of CORBA: A Framework for Distributed Applications in Defence

Executive Summary

Distributed object computing is a promising technology that is fast becoming the dominant computing paradigm of the future. Inevitably, a critical part of software development is integrating present and future software, so that legacy systems can still remain fully operational in new environments. Using an object-oriented approach, the Object Management Group (OMG) defines an Object Management Architecture (OMA) for the support of interoperable applications across heterogeneous hardware platforms and operating systems. By adopting interface and protocol specifications, the OMA is expected to provide an underlying model for all software components, embracing those which were previously developed or which are not necessarily object-oriented.

The Common Object Request Broker Architecture (CORBA) is the communication heart of the OMG OMA, which provides a framework for flexible and transparent communication between distributed objects in heterogeneous computing environments. CORBA is a well-established and widely adopted standard that specifies the attached components and how they interoperate. Essentially, distributed CORBA objects provide scalable and flexible solutions for heterogeneous environments and for the Internet and intranets. The adoption of this approach eases software development by allowing interaction between reusable components through well-defined interfaces.

Owing to the increasing dominance of joint military operations in the future, the interoperability of military communications and information systems within and across individual Services (Navy, Army, and Air Force) will be more prominent. Applying CORBA technology to C4I problems in the military environment provides simple integration of these systems. Such integration greatly increases the value of information that spread across various defence applications and different computer platforms, thereby fulfilling the operational requirements of modern battlefield interoperability.

This report is intended to serve as a tutorial on CORBA, covering from the OMA to each component of CORBA. The OMG is also striving, at a high level of abstraction, to define various services and facilities necessary for distributed object computing. These components will provide fundamental object interfaces necessary for building object frameworks towards specific application domains such as telecommunications, medical systems, finance, manufacturing, and C4I systems.

Authors

T. A. Au
Communications Division

Andrew Au is a Research Scientist in Network Integration Group with interests in traffic control and performance guarantees for high speed networks. He has been investigating ATM signalling and communications issues in distributed computing environments.

Contents

1. Introduction

The prevalence of computers in routine business functions has created a heterogeneous information-processing environment, embracing vast networks of autonomous and distributed computing resources. Clearly, there is a growing need for technology to flexibly coordinate these diverse computing resources to fully integrate distributed systems. These efforts are expected to facilitate portability of applications and interoperability of systems and networks, in support of challenging new information processing requirements.

In recent years, distributed computing has made significant advances to support objects distributed across a network, showing dominance of the object-oriented paradigm. An object is a unique instance of a data structure (abstract data type) encapsulated with a set of routines, called methods which operate on that data. This approach entails the transparent distribution of applications across networks of heterogeneous computers from different vendors. For large organisations such as defence, legacy systems pose a serious problem because these systems are mission-critical and must remain fully operational at all times even in new computing environments.

One of the main problems facing distributed computing is software component integration. To achieve this integration, the Object Management Group (OMG) has undertaken definition of a distributed object computing platform for inter-communication of application objects in widely distributed, heterogeneous environments. With over 800 member companies, the OMG is an international industry consortium formed to develop, adopt, and promote standards for distributed object computing. The OMG Object Management Architecture (OMA) aims to define at a high level of abstraction, various facilities necessary for distributed object computing. Within this architectural framework, the technology adopted for Object Request Brokers (ORBs) is known as the Common Object Request Broker Architecture (CORBA) for transparent communication between application objects. Indeed, this is an OMG specification based on the standard interface definition between OMG-compliant objects.

Military communications and information systems have been characterised by the development of expensive, purpose-built and non-interoperable systems. Owing to the increasingly dominance of joint military operations in the future, the interoperability of these systems within and across Services will be more prominent. Object technology, particularly the CORBA, offers several benefits for military systems including support for application diversity, technology insertion, system evolution and distribution. The integration of interoperable military systems greatly increases the value of information spread across various defence applications over different computer platforms, thereby fulfilling the operational requirements of modern battlefield interoperability.

This report is intended to provide an overview of OMA, and a snapshot of CORBA, in achieving interoperability across platforms and applications. Section 2 discusses increasing degrees of interoperability from basic interconnectivity to business collaboration. Section 3 introduces the OMG OMA. Section 4 explains the CORBA Object Model, and Section 5 provides an overview of CORBA. Section 6 describes the OMG Interface Definition Language (IDL) that bridges diverse programming languages, operating systems, networks and object systems. CORBA supports both static and dynamic method invocations, as discussed in Sections 7 and 8, respectively. Section 9 presents the Interface Repository (IR) that stores the interface specifications of each object on the CORBA object bus. The interface to the ORB functions is described in Section 10, and the Basic Object Adaptor (BOA) is discussed in Section 11. A general ORB interoperability architecture is introduced in Section 12 for the support of distributed objects across heterogeneous ORBs. Finally, concluding remarks are drawn in Section 13.

For an introductory overview of the CORBA, readers may choose to skip Sections 6-11, which are involved in more technical detail.

2. Increasing Degrees of Interoperability

There are many degrees of interoperability in a computing environment. Basic interconnectivity only allows simple data transfer, whereas application-level interoperability enables applications running in any environment to exchange information and perform processing, even if they were developed at different times by different developers. To this end, we identify three phases which are predominantly involved in creating integrated, large-scale, distributed information systems.

1. Network Interconnectivity

   To guarantee basic communication, computing resources are first interconnected to exchange messages. Based on the prevalent client-server model, a distributed system is organised as a number of distributed server processes that offer various services to client processes across a network [1]. By means of interprocess communication mechanisms such as remote procedure calls (RPCs), servers provide clients with access to general system-defined services. Examples are file-storage, printing, authentication, and naming services.

2. Software Interoperability

   A more ambitious goal is to execute tasks jointly among interoperable computing resources. Such interoperability involves intricate interactions through the use of programming capabilities such as an RPC mechanism extended with a data-translation facility [1]. Examples are the integration of heterogeneous information in advanced multimedia applications, and information storage in integrated repositories. Although the basic client-server model does support a certain level of

interoperability, the complexity of migrating from locally distributed systems to more global systems demands new tools and techniques. In addition, greater interoperability is required to allow various programming languages and development tools to work together to reuse functions across platforms.

3. Business Collaboration

   Business collaboration goes beyond software interoperability, simply by expanding service boundaries of the software component infrastructure. This infrastructure provides application-level collaboration in the form of application frameworks, allowing developers to customise implementation methods based on business requirements [2]. At this application level, software components are created to behave like business objects[1]. They collaborate, more than just interoperate, at the semantic level to accomplish a business process. For example in a car reservation system, four business objects can be defined to represent customer, invoice, car, and car lot with some agreed-upon semantics for communicating with each other to perform business transactions.

In recent years, the trend towards higher levels of interoperability is evidenced by the rapid growth in middleware technology. This category of software mediates between an application program and a network so that the specifics of the operating environment are isolated from the application. In addition, middleware provides interfaces between clients and servers by managing the interaction between disparate applications across heterogeneous computing platforms. An example of a middleware program is the Object Management Group (OMG)'s Common Object Request Broker Architecture (CORBA) that manages communication between objects.

Through the development of standards for distributed object computing, the OMG attempts to provide a common architectural framework for portability and interoperability in heterogeneous computing environments. By modelling a distributed system as a collection of interacting objects, distributed components can communicate with each other only using messages addressed to well-defined interfaces. This object concept produces a natural model for integrating distributed computing resources, which can be effectively applied to both distributed computing and telecommunications environments. As object-oriented computing becomes more mature and moves into the mainstream, the work of the OMG will increase in importance.

[1] A business object is a piece of information which is of interest and recognisable to the business. A set of different business objects working together is able to define a business enterprise. For instance, one business object represents the accounts receivable while another business object represents manufacturing.

3. The Object Management Architecture

The OMG has developed is a high-level conceptual model of a complete distributed environment, called the Object Management Architecture (OMA). This model specifies an architectural framework of distributed objects that helps reduce the complexity, lower the costs, and hasten the introduction of new software applications. Every component in the architecture is defined in terms of an object-oriented interface. Only through these interfaces can an object request services from any other object. The OMA is supported by detailed interface specifications that drive the industry towards interoperable, reusable, portable software components based on open, standard object-oriented interfaces. Hence, this architecture becomes an underlying model for all software components, including those which were previously developed or which are not necessarily object-oriented.

The OMG Object Model provides an organised presentation of object concepts and terminology, in which common object semantics are defined in a standard implementation-independent way. These semantics specify the externally visible interfaces that are used to interact with object state and object behaviour. Clients issue requests to object services only through these well-defined interfaces, specified in the OMG Interface Definition Language (IDL). The request carries information including an operation, the object reference of the service object, and parameters, if any. Typically, the implementation and location of each object are hidden from the requesting client, thus ensuring interoperability between software components and portability of applications. In each potential request, an interface to the target object is a description of a set of possible operations that are specified in the OMG IDL.

The OMA Reference Model identifies and characterises components, interfaces, and protocols, but does not in itself define them in detail. A particular computing or business problem space is partitioned into practical, high-level architectural components that can be addressed by various technologies. This model therefore provides the means to build interoperable software systems distributed across all hardware and software environments.

3.1 OMA Component Definitions

The OMA Reference Model identifies five components in heterogeneous environments: Object Request Broker, Object Services, Common Facilities, Domain Interfaces, and Application Objects. The OMA can also be viewed as three major segments consisting of these five critical components:

1. Application Oriented Segment - Application Objects and Common Facilities are solution-specific components, which are located closest to the end user.

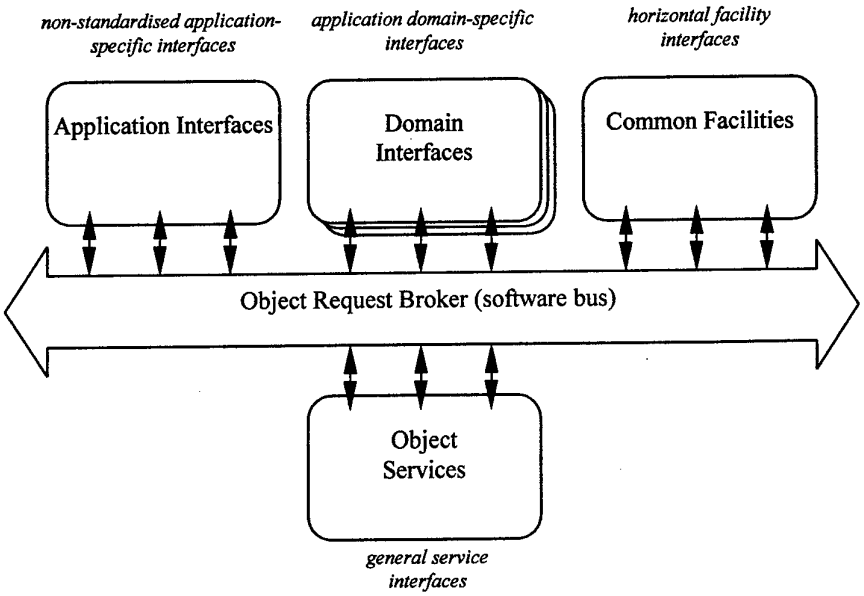


Figure 1 : Object Management Architecture Reference Model

2. System Oriented Segment - the underlying infrastructure of distributed object computing environments comprises of Object Request Broker (the communications heart), and Object Services.
3. Vertical Market Oriented Segment - Domain Interfaces provide vertical extensions to specific applications or domains, applicable for a variety of industries.

Figure 1 illustrates the Object Request Broker component and its interactions with the other four categories of object interfaces. These five OMA components are defined as follows:

- Object Request Broker (ORB)

  The ORB is the "broker" which lets applications request use of another object without knowing where that other object is located on the system or network. It provides the basic object interaction capabilities which are necessary for any of the components to communicate, independent of the specific platforms and implementation techniques. The ORB finds the requested object, wherever it is located, and passes the requested information to the requested object. It also passes information back to the requester, as necessary. The ORB component guarantees portability and interoperability of objects over a network of heterogeneous systems. The ORB is commercially referred to as the Common Object Request Broker Architecture (CORBA) in which the programming interfaces to the ORB component are defined.

- Object Services

  The Object Services component standardises the life cycle management of objects which are low-level system type services necessary for developing applications. Functions are provided to create objects, to control access to objects, to keep track of relocated objects and to consistently maintain the relationship between groups of objects. These general purpose services are used by many distributed object programs to provide for application consistency and to increase programmer productivity. They are indeed the basic building blocks for distributed object applications, from which higher level facilities and object frameworks can be constructed for interoperability across multiple platform environments. Adopted OMG Object Services are collectively called CORBAservices. Typical examples are the naming service, and the trading service for the discovery of other available services. Other services include life cycle management, security, transactions, and event notification. Specifications for Object Services are contained in *CORBAservices: Common Object Services Specification.*

- Common Facilities

  The Common Facilities component provides a set of generic application functions that can be configured to the requirements of a specific configuration, leading to uniformity in generic operations and to better options for end users for configuring their working environments. These are higher level services, which are semantically closer to the application objects. For example, printing, document management, database, and electronic mail facilities are readily usable by many applications. The availability of such capabilities allows applications to be created quickly in a portable and interoperable way. Adopted OMG Common Facilities are collectively called CORBAfacilities in which an OpenDoc[2]-based Distributed Document Component Facility (DDCF) is included. The OMG intends to collect information about Common Facilities in *CORBAfacilities: Common Facilities Architecture.*

- Domain Interfaces

  These domain-specific interfaces are oriented towards many separate applications domains. Examples are finance, manufacturing, telecommunications and medical domains.

- Application Objects

  To perform specific tasks for users, new classes of Application Objects can be built by modifying existing classes through generalisation or specialisation provided by Object Services. Application Objects are actually business objects and applications. These are the ultimate consumers of the CORBA infrastructure. A particular application can be constructed from a large number of basic object classes, partly

[2] OpenDoc is a compound document architecture that enables embedding of features from different application programs into a single working document.

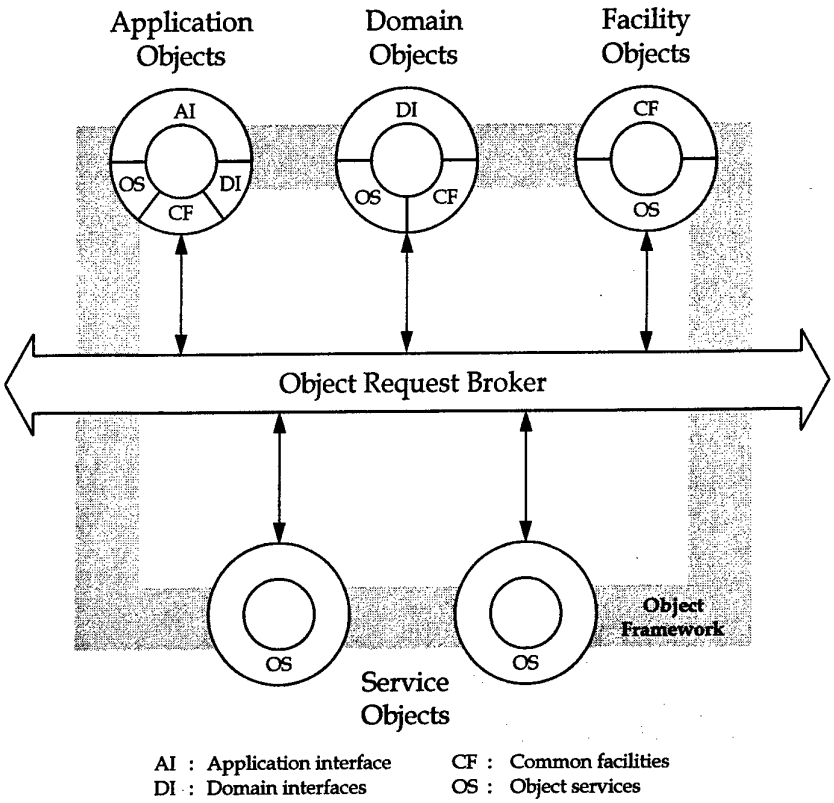


Figure 2 : OMA Reference Model Interface Usage

specific for the application, partly from the set of Common Facilities, thereby improving developer productivity and configuration flexibility.

3.2 Object Frameworks

Object Frameworks are groups of higher level components that interact to provide functionality of direct interest to end-users in particular application or technology domains [3]. These are collections of cooperating objects typically oriented towards domains such as telecommunications, health care, finance, and manufacturing. The cooperating objects can also be categorised into application, domain, facility, and service objects. Each object or component in an object framework supports some combination of application, domain, common facility, and object services interfaces, as illustrated in Figure 2. Some components support all types of interfaces, while others support a subset of these interfaces. In general, service objects support object services interfaces, whereas facility objects support common facilities interfaces and potentially inherited object services interfaces. Domain objects support domain interfaces, and potentially inherited common facility and object services interfaces. Likewise, application objects support application interfaces and potentially all the other interfaces.

The basic client-server model is used to coordinate the interactions between two objects. In order to provide the overall functionality of the object framework, objects are able to make requests to all other objects on a peer-to-peer basis. Each component uses the ORB to communicate with other components through the supported interfaces. Depending on the occasion, an object can therefore act as either a client or a server.

4. The CORBA Object Model

The OMG Object Model in the Object Management Architecture (OMA) is an abstract object model that is not directly realised by any particular technology. The CORBA Object Model, on the other hand, is a concrete object model based on the OMG Object Model, which provides the underlying definitions for a particular technology. This model encompasses more detailed specifications defined from the abstract concepts, in particular, the interactions between clients and servers.

4.1 Requests

A client of a service is any entity capable of requesting the service. A request is an event that associates a set of information at a particular time, consisting of an operation, a target object, zero or more (actual) parameters, and an optional request context. In addition, a request form is supported as a description or pattern that can be evaluated or performed multiple times to issue requests.

A client may request one or more services from an object. For the purpose of identifying the request, an object may be identified by a value called an object name. An object reference, however, reliably identifies a particular object that may be denoted by multiple, distinct object references. Additional information about the request may be provided by a request context. A request causes a service to be performed on behalf of the client, and results, if any, may be returned to the client. An abnormal condition may generate an exception to the client, carrying additional return parameters particular to that exception.

4.2 Types

A signature defines the types of the parameters for a given operation. These types are used to restrict a possible parameter or to characterise a possible result. The set of values that satisfy a type is called the extension of the type. A type whose members are objects is an object type. Values in a request are restricted to values that satisfy these type constraints. Legal values of the OMG type hierarchy are shown in Figure 3. The set of basic values includes various forms of integers, floating points, characters, booleans, and any. Constructed values are structured types such as records (called

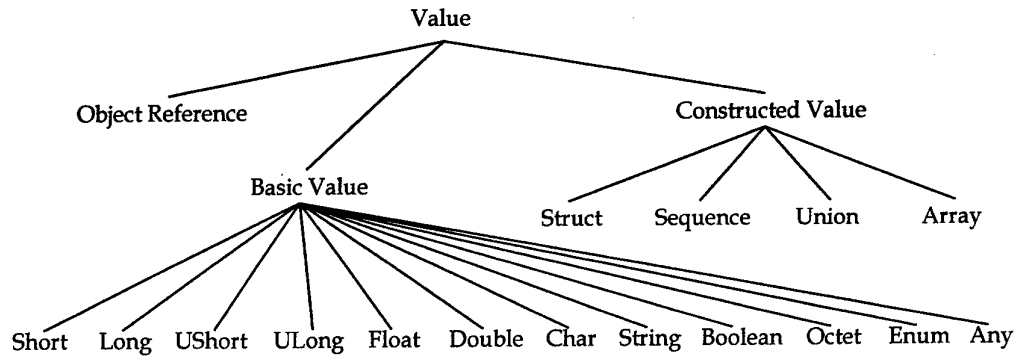


Figure 3 : Legal Values of OMG Type Hierarchy

structs), unions, arrays, and sequences. The type *any* is used to represent any possible basic or constructed type.

4.3 Interfaces

An interface is a description of a set of possible operations that a client may request of a target object. Interfaces are defined in OMG Interface Definition Language (IDL). An object is able to support multiple interfaces through interface inheritance. An interface may have attributes. A pair of accessor functions are declared: one to retrieve, and the other to set the value of an attribute. However, only the retrieval accessor function is defined in the case of read-only attribute.

4.4 Operations

An operation denotes a service that can be requested. An operation is an entity identified by an operation identifier. The signature of an operation describes the legitimate values of request parameters and returned results, which can be represented as

```
[oneway] <op_type_spec> <identifier> (param 1, …, paramL)
   [raises(except1, …, exceptN)] [context(name1, …, nameM)]
```

Two styles of execution semantics are defined: *at-most-once* and *oneway*. *At-most-once* is the default semantics, indicating that if the operation successfully returns results, it was performed exactly once, and if an exception is returned, then the operation was executed either once or not at all. *Oneway* specifies that best-effort semantics are expected of requests for this operation. The client does not wait until the completion of the operation and no result is returned.

The `<op_type_spec>` defines the type of return result and the `<identifier>` specifies an operation name in the interface. The mode of each parameter can be either *in*, *out*, or *inout*, according to the direction in which the information flows. The set of

allowed values is defined in the type of each parameter. The optional `raises` expression indicates that user-defined exceptions can be signalled to terminate the request. The optional `context` expression provides additional request context information to the object implementation on an operation-specific basis.

4.5 Object Implementation

The object implementation carries out the computational activities needed to effect the behaviour of requested services. Essentially, the CORBA implementation model consists of an execution model and a construction model.

The execution model describes how services are performed. A requested service is performed by executing a code segment, or called a method, that operates upon some data. The data represents a component of the state of the computational system, which may be changed as a result of the execution.

The construction model describes mechanisms for realising behaviour of requests. These mechanisms include definitions of object state and methods, and definitions of how method dispatch is performed. In particular, this model describes how object implementations are constructed, including the information needed to create an object so as to provide an appropriate set of services.

5. Common Object Request Broker Architecture

The Common Object Request Broker Architecture (CORBA) is a first step by the Object Management Group towards achieving application portability and interoperability across heterogeneous computing platforms. It is a standard for the development and deployment of applications in distributed, heterogeneous environments. CORBA 1.1 was first introduced in 1991. In CORBA 2.0, adopted in December 1994, true interoperability is defined by specifying how ORBs from different vendors can interoperate.

A client makes a request for a service provided by an object implementation through the Object Request Broker (ORB). The client is not required to care about the location of the object implementation, its programming language, or any other details. To support portability and interoperability, the ORB is responsible for all of the mechanisms required to locate the object implementation, and to prepare the object implementation to receive the request and its data. Figure 4 shows the main components of the ORB architecture and their interconnections.

The client makes a request using either the Dynamic Invocation Interface (DII) or an OMG IDL stub. The client can also directly interact with the ORB through the ORB Interface for some functions. Similarly, the object implementation receives a request

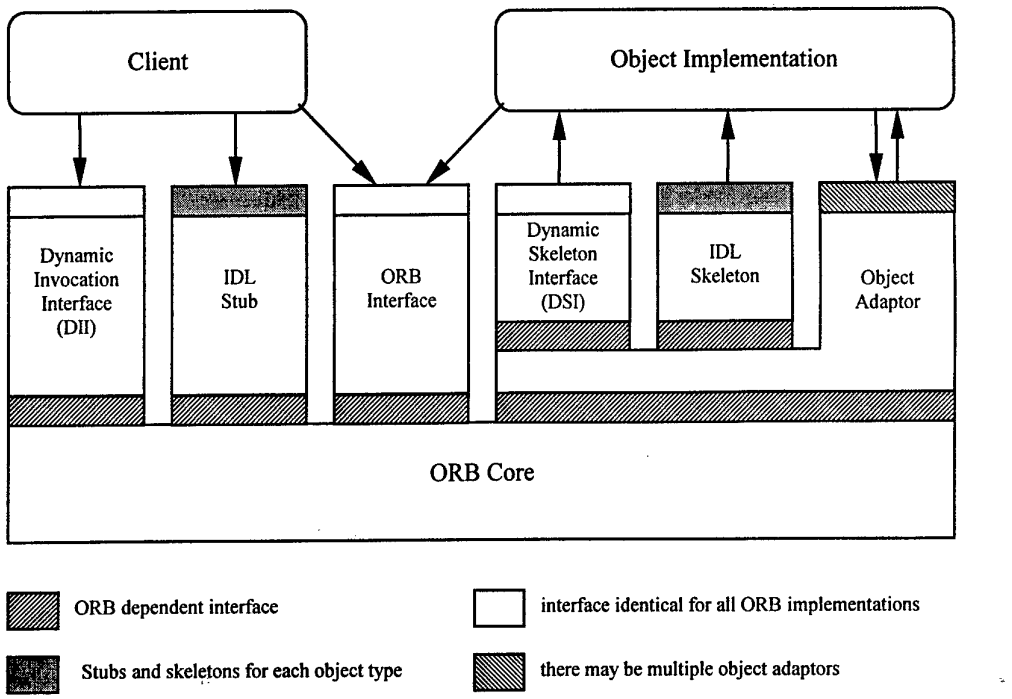


Figure 4 : The Structure of Object Request Broker Interfaces

either through the OMG IDL skeleton or through a Dynamic Skeleton Interface (DSI). The object implementation may call the Object Adaptor and the ORB Interface for the ORB services.

Having access to the object reference as a handle to the object implementation, the client initiates the request by statically calling the IDL stub, or by constructing the request dynamically through the Interface Repository. An OMG IDL stub is the specific stub depending on the interface of the target object, whereas the DII interface is independent of the interface of the target object, which therefore can be used even without thorough knowledge of that interface. Nevertheless, both the static and dynamic invocation methods generate the same request semantics to the object implementation.

The ORB locates the target object, transmits parameters and transfers control to the object implementation through an IDL skeleton or a dynamic skeleton. Defining the interface of an object in IDL generates an IDL skeleton that is specific to the interface and the object adaptor. The object implementation information provided at installation time is stored in the Implementation Repository, which is available for use during request delivery.

Figure 5 shows how interface and implementation information is made available to clients and object implementations. The interface is defined in OMG IDL and/or in the

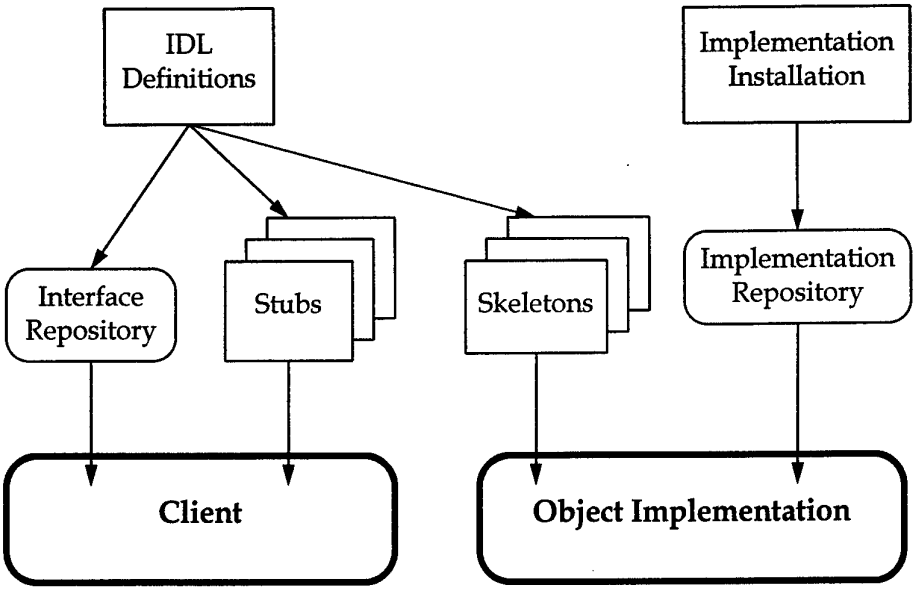


Figure 5 : Interface and Implementation Repositories

Interface Repository. The client stubs and the object implementation are generated from the IDL definition of the interface.

5.1 Object Request Broker

CORBA is designed to support different object mechanisms by structuring the ORB with components above the ORB core. The ORB core is actually the ORB component that moves a request from a client to the appropriate adaptor for the target object, thus providing the basic representation of objects and communication of requests. Multiple ORB implementations may have different representations for object references and different means of performing invocations. The ORB core is defined by its interfaces that can mask the differences between ORB implementations. ORB cores, together with the IDL compilers, repositories, and various object adaptors, provide a set of services to clients and implementations of objects that have different properties and qualities. While working together, different ORB implementations must be able to distinguish their object references.

5.1.1 Example ORBs

A number of ORB implementations are possible within the specifications of CORBA. A particular ORB, however, might support multiple options and protocols for communication.

Client- and implementation-resident ORBs are ORBs that are implemented in routines resident in the clients and implementations. These routines collectively provide the ORB functionality to each client or implementation process. To facilitate requests, the

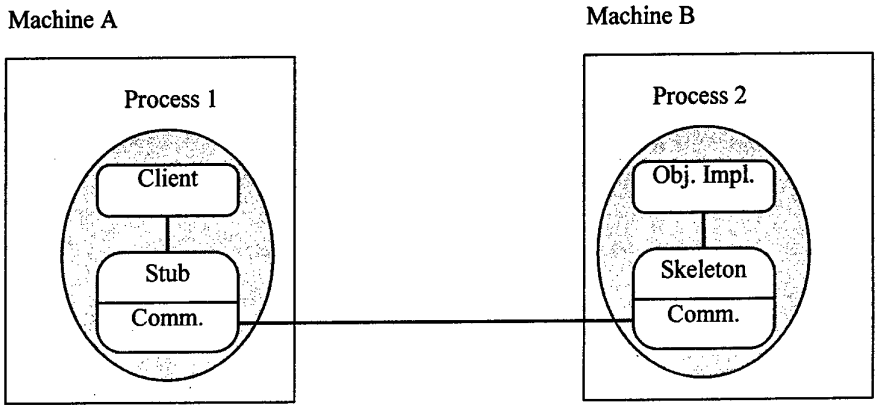


Figure 6 : Client- and Implementation-resident ORB [4].

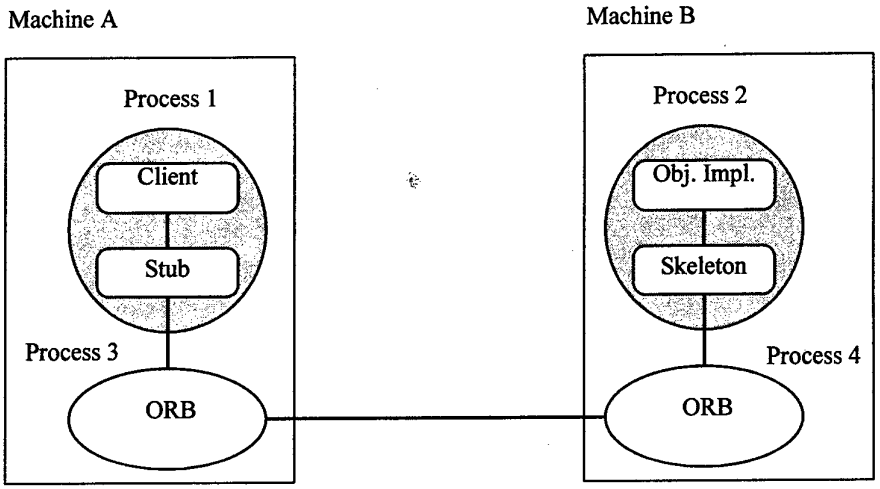


Figure 7 : Server-based ORB [4]

stubs in the client either use a location-transparent inter-process communication mechanism or directly access a location service to establish communication with the object implementations. This is illustrated in Figure 6.

A server-based ORB is one that centralises the management of an ORB within one or more servers. All clients and implementations communicate via the interaction with these servers which facilitate the routing of requests from clients to implementations. Figure 7 shows a server-based ORB.

In a system-based ORB, the ORB functionality is provided as a basic service of the underlying operating system to enhance security, robustness, and performance. A variety of optimisations can be implemented because the operating system knows the

location and structure of clients and implementations, such as avoiding marshalling[3] when both are on the same machine.

A library-based ORB keeps implementations of light-weight objects in a library. In this case, the stubs are the actual methods. It is assumed that a client program can get access to the data for the objects and that the implementation trusts the client not to corrupt the data.

5.2 Object References

An object reference is an opaque representation to specify an object within an ORB. Both clients and object implementations use object references upon which requests are issued. Object references are uniform only within an ORB implementation. The representation of an object reference handed to a client is only valid for the lifetime of that client. Different ORB implementations can provide different representations of object references. Nevertheless, all ORBs must provide the same language mapping to an object reference for a particular programming language, independent of a particular ORB. Such a mapping insulates both clients and object implementations from the actual representation.

Whenever an object reference is passed across ORBs, the originating ORB must create an Interoperable Object Reference (IOR) across object reference domain boundaries. This IOR data structure comprises a collection of object-specific tagged profiles. These profiles encapsulate all the basic information that the protocol in a foreign ORB needs to identify an object.

5.3 Clients

A client object requests service from another object using an object reference, an operation name, and a set of parameters. An object reference serves as a handle of the target object on which the operation is requested. A client makes a request from within the application code which uses object-type-specific stubs as library routines in the program. The client passes a language-specific data type as an object reference to the stub routines to initiate an invocation.

Once the stub routine is called to perform the invocation, the stub interacts with the ORB by mapping the object reference for the target object to the object reference representation in the ORB implementation. This is shown in Figure 8. The ORB is responsible for locating the object implementation, and for managing the transfer of the

[3] Marshalling is the process of packing one or more items of data into a message buffer, prior to transmitting that message buffer over a communication channel. The packing process not only collects together values which may be stored in non-consecutive memory locations but also converts data of different types into a standard representation agreed with the recipient of the message.

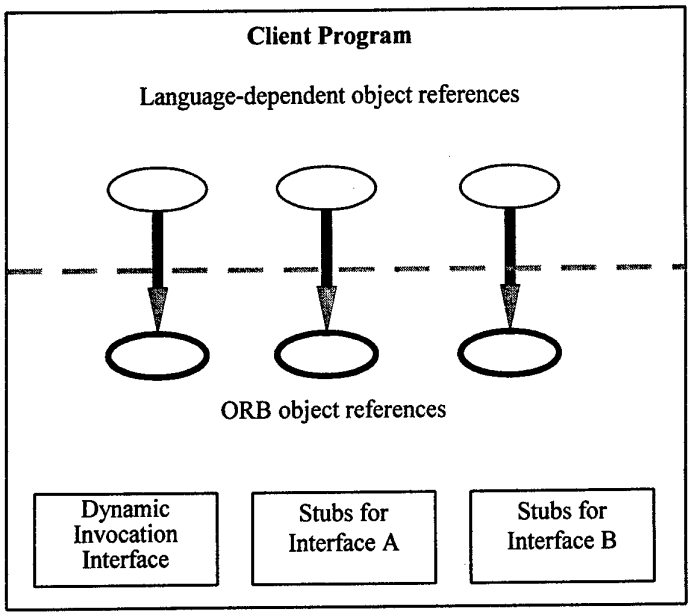


Figure 8 : Structure of a Typical Client

request and results, if any, between the client and the object implementation. An exception is returned in the event of an error or an incomplete invocation.

When the target object was undefined at the compile time of the client object, it is possible that the stubs of the object implementations are not available to the client code. In this case, the client program provides additional information to name the type of the object and the method being invoked. Invocations on target objects are performed via the Dynamic Invocation Interface. The construction of this request however involves a sequence of calls to specify the parameters and to initiate the invocation.

5.4 Client Stubs

On behalf of clients, stubs are local proxies for potentially remote objects, making remote invocation look similar to local invocation by hiding the details of the underlying ORB from the application programmers. Clients issue requests from within their host programming languages through stubs, invoking remote operations defined as part of the target object interface. A mapping from an IDL definition generates a stub for each interface within the native programming language. Object-oriented programming languages may simply view CORBA objects as programming language objects. For the mapping of a non-object-oriented language, a programming interface to a stub routine is required for each interface type, providing access to a particular IDL-defined operation on a particular object. As the client stubs make calls on the rest of the ORB using interfaces that are private to the particular ORB core, different ORBs may require correspondingly different stubs. In this case, it is necessary for the ORB and

language mapping to cooperate to associate the correct stubs with the particular object reference.

5.5 Dynamic Invocation Interface

The Dynamic Invocation Interface allows the dynamic construction of object invocations. A client can directly specify the object to be invoked, the operation to be performed, and the set of parameters for the operation through a call or a sequence of calls. The Dynamic Invocation Interface is common to all objects and all operations. It does not make use of the stub routines generated for each operation in each interface. Information regarding the parameters and the operation itself is usually acquired from the Interface Repository.

5.6 Interface Repository

The Interface Repository maintains persistent IDL definitions that are available at run time. This information is used by the Dynamic Invocation Interface to issue requests on object interfaces that were unknown when the client program was compiled. The ORB itself also makes use of the Interface Repository services to perform requests. The Interface Repository provides a persistent store for additional information associated with interfaces to ORB objects. Examples are annotations and debugging information, libraries of stubs or skeletons, and routines that can format or browse particular kinds of objects.

5.7 Object Implementations

The actual state and behaviour of an object are encapsulated in an object implementation. In the CORBA specification, only the necessary mechanisms for invoking operations are defined. The implementation of an object is free to specify its own procedures for activating and deactivating objects, access control, as well as keeping persistent objects.

An object implementation interacts with an ORB to obtain services, as shown in Figure 9. Primarily, an object adaptor provides the object implementation an interface to ORB services such as establishing the identity of the object implementation, and creating new objects. When an invocation occurs, the ORB core, the object adaptor, and a skeleton arrange to make the call to the appropriate method of the implementation. The method can be called statically from the interface skeleton. Object references are passed for which the call was made, together with parameters identifying the object being invoked, and the particular method. Alternatively, a dynamic skeleton makes the invocation possible even if static skeletons for each object interface type have not been compiled. When the method is finished, output parameters or exception results are returned back to the client.

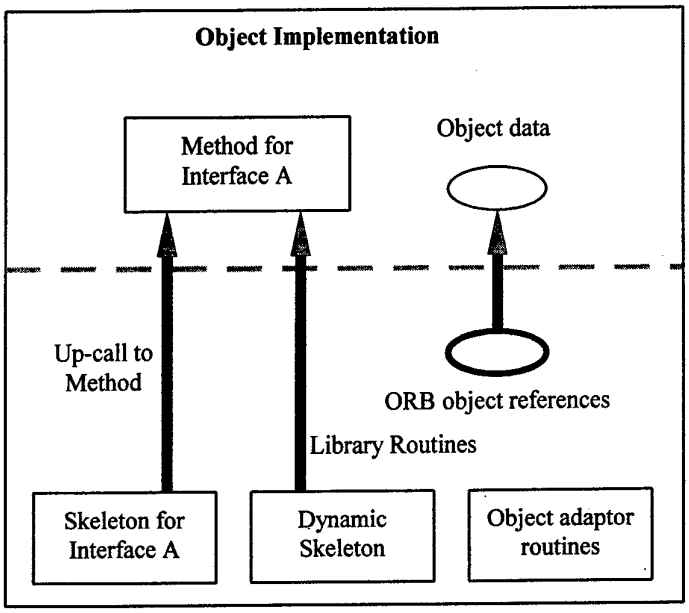


Figure 9 : Structure of a Typical Object Implementation

The ORB may be notified of the creation of a new object so that the ORB is able to locate the implementation for that object. The implementation also registers itself as implementing objects of a particular interface, and specifies how to start up the implementation if it is not running.

5.8 Implementation Skeleton

Implementation skeletons are the structures, which actually invoke the methods implemented as part of the object implementation. They are created from the IDL definitions for each interface within a programming language. The skeleton is filled with actual code that will be invoked when a request is received. The ORB calls a particular operation through the specific skeleton. However, it is possible to write an object adaptor that does not use skeletons to invoke implementation methods.

5.9 Dynamic Skeleton Interface

A Dynamic Skeleton Interface allows dynamic handling of object invocations to an object implementation without the use of an implementation skeleton. Information such as the operation name and parameters are determined by means of purely static knowledge or dynamic knowledge through an Interface Repository.

The implementation code must provide descriptions of all the operation parameters to the ORB, and the ORB provides the values of any input parameters for use in performing the operation. The implementation code returns the values of any output parameters, or an exception, to the ORB after performing the operation. The nature of

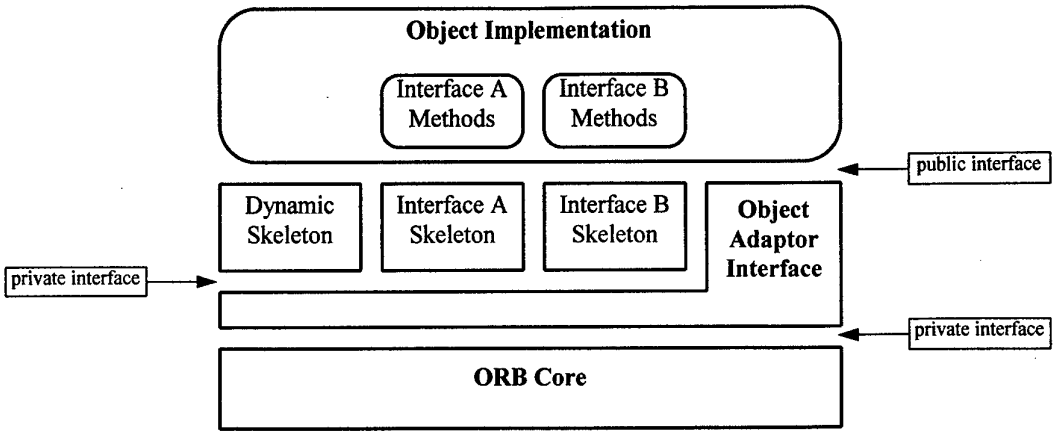


Figure 10 : Structure of a Typical Object Adaptor

the Dynamic Skeleton Interface may vary substantially from one programming language mapping or object adaptor to another.

5.10 Implementation Repository

The Implementation Repository contains information that allows the ORB to locate and activate the object implementations necessary to fulfil a request. The Implementation Repository also stores additional information associated with the object implementations. Examples are debugging information, administrative control, resource allocation, and security. The Implementation Repository is specific to an operating environment, because it is used in the construction and activation of the object implementations.

5.11 Object Adaptors

An object implementation primarily accesses ORB services through an object adaptor. When a request comes in from the ORB core, the object adaptor provides the required services and helps deliver the request to the skeleton for method invocation. An object adaptor is implicitly involved in invocation of the methods through the skeletons. It publishes a public interface to the object implementation and a private interface to the interface skeleton. An object adaptor itself makes use of a private ORB-dependent interface, as shown in Figure 10.

The functions that an object adaptor performs include: generation and interpretation of object references, method invocation, security of interactions, object and implementation activation and deactivation, mapping object references to the corresponding object implementations, and registration of implementations.

A number of object adaptors can be envisioned for different groups of object implementations based on object granularities, lifetimes, policies, implementation

styles, and other properties. Each object adaptor is typically designed to support a range of object implementations.

Object adaptors are primary service providers for object implementations. Object adaptors can be specialised and only a Basic Object Adaptor (BOA) is defined in the CORBA specification. The BOA can be used for most ORB objects with conventional implementations. It provides a small amount of persistent storage for each object, which can be used as a name or identifier for other storage, for access control lists, or other object properties. Examples of special-purpose adaptors are the library object adaptor for objects that have library implementations, and the object-oriented database adaptor to provide access to the objects in an object-oriented database.

5.12 ORB Interface

The ORB Interface is identical for all ORB implementations, providing operations common across all objects. This interface is mapped to the host programming languages used by the clients and the object implementations.

6. OMG Interface Definition Language

The OMG Interface Definition Language (IDL) is used to define interfaces through which a client is informed of the services supported by an object implementation. OMG IDL is purely a descriptive language so that an interface is defined in a language-neutral way. It enables potential clients to identify the available operations and their invocation methods for a particular object implementation. An interface definition written in OMG IDL completely defines the interface and fully specifies the operation signatures, embracing a set of named operations and the parameters to those operations. Hence in CORBA, the OMG IDL definitions are used to create the stubs and skeletons, as well as to populate the Interface Repository.

To specify the parameter types and return types for operations, OMG IDL provides a set of data types that are similar to those found in a number of programming languages. It supports built-in data types such as long, short, float, double, char and boolean. The sizes, byte ordering and interpretation of all these basic types are precisely defined to ensure interoperability across heterogeneous platforms. OMG IDL also supports constructed types such as struct and discriminated union, and template types such as string and sequence whose exact characteristics are defined at declaration time. This IDL type hierarchy is covered in Section 4, where Figure 3 illustrates all the legal values. These are used in operation declarations to define argument types and return types. Operations are also specified in interface declarations to identify the services provided by objects.

More importantly, interfaces are used as object reference types to describe CORBA objects. An operation can take object references as arguments and return the appropriate object references. In addition, OMG IDL provides interface inheritance, allowing derived interfaces to inherit the operations and types defined in the base interfaces. All IDL interfaces are implicitly derived from a root interface called Object defined in the CORBA module, which provides services common to all ORB objects.

CORBA objects, expressed in OMG IDL, are mapped into particular programming languages or object systems according to the specifications of the object interfaces. A client uses the IDL interface specifications, but the actual call is done in the native programming language, as a result of mapping of the IDL interfaces to the programming language. As expected, the mapping of OMG IDL to a particular programming language is the same for all ORB implementations. This includes definition of the language-specific data types and procedure interfaces to access objects through the ORB.

OMG IDL obeys the same lexical rules as C++. It has similar syntax and the grammar is actually a subset of the C++ grammar with additions necessary for distributed invocations. The C++ concept of class roughly corresponds to the concept of an IDL interface.

6.1 IDL Specifications

An entire IDL file forms a naming scope[4] within which one or more IDL specifications are contained. An IDL specification consists of one or more type definitions, constant definitions, exception definitions, interface definitions, or module definitions. Identifiers of types, constants, attributes, operations, and exceptions can only be defined once in a naming scope. However, identifiers can be redefined in nested scopes.

An IDL module provides a namespace to group a set of interfaces, allowing for scoping of definition names to prevent name clashes. An interface defines a set of operations (or methods) without the actual implementation, which a client can invoke on an object. One or more exceptions can be declared to indicate an operation failure. An interface may have attributes for which the implementation automatically creates `get` and `set` operations. An interface can be derived from one or more interfaces, thus supporting multiple interface inheritance. An operation signature designates the parameters and the results that the method returns. In particular, the mode of a parameter indicates whether the value is passed from client to server (*in*), from server to client (*out*), or both (*inout*).

[4] The scope of an identifier is the region of a program source that usually extends from the place where it is declared to the end of the smallest enclosing block. If a name is not resolved within a particular scope, it is searched for in successively expanding outer scopes.

```
module MyAnimals                                    <-- Defines a naming context
{
     // Class Definition of Dog
     interface Dog : Pet, Animal                    <-- Defines a CORBA class
     {
          attribute integer age;
          exception NotInterested {string explanation};

          void Bark(in short how_long)              <-- Defines a method
               raises (NotInterested);

          void Sit(in string where)
               raises (NotInterested);

          void Growl(in string at_whom)
               raises (NotInterested);
     }

     // Class Definition of Cat
     interface Cat : Animal
     {
          void Eat();
          void HereKitty();
          void Bye();
     }
}    // End MyAnimals
```

Figure 11 : IDL Module MyAnimals with Two Interfaces, Dog and Cat [2]

An interface definition is composed of a header and a body. The header specifies the name of the interface and an optional inheritance structure. The body contains declarations of constants, types, exceptions, attributes, and operations. Figure 11 shows an example of an IDL file [2] with two interfaces, `Dog` and `Cat`, defined in a module called `MyAnimals`. Interface `Dog` is derived from two base interfaces, `Pet` and `Animal`. `Dog` includes a new declaration of attribute called `age`, for which the implementation automatically provides `get` and `set` methods. Three methods are supported in `Dog`: `Bark`, `Sit`, and `Growl`, but an exception may be raised when the dog is not in the mood to obey. Likewise, Interface `Cat` is derived from `Animal`, and supports three methods: `Eat`, `HereKitty`, and `Bye`.

6.2 Programming Language Mappings

Interfaces are only defined in the OMG IDL specifications. Actual client and object implementations still have to be done in programming languages. Language mappings determine how OMG IDL features are mapped to the facilities of a given programming language. Hence, a procedure or function that a client invokes from within its programming language is mapped to a corresponding IDL operation. A request is then initiated to the object implementation via the ORB and the object adaptor. Subsequently, the IDL skeleton calls the requested procedure of the object implementation in its native programming language.

Each programming language mapping includes a static mapping of interfaces, and a mapping of the Dynamic Invocation Interface (DII). The static mapping translates IDL specifications and CORBA-defined interfaces to the programming language stubs. This stub declaration is then compiled in the programming code of a client to originate a particular request to the object implementation. This static mechanism requires pre-compilation. The DII however allows requests to be dynamically built even before the interfaces for the operations are defined. Essentially, all operations are viewed as common constructs that have a name, a list of parameters, and an object reference. The DII thus provides a dynamic mechanism by which any operation can be invoked.

Language mappings provide means of expressing IDL data types, constants, object references, operations, attributes, and exceptions, so that the interfaces to the ORB are characterised in different programming languages. A language mapping involves definition of the language-specific data types and procedure interfaces to access objects through the ORB. In any case, a particular mapping of IDL to a programming language should have similar structure with the same set of features for all ORB implementations. Different programming languages may access CORBA objects in different ways. Object-oriented languages may perceive CORBA objects as their own programming language objects, whereas non-object-oriented languages may prefer to hide the exact ORB representation.

7. Static Method Invocation

As mentioned above, CORBA supports two types of client-server invocations: static, and dynamic method invocations. The static invocation is a very natural form of programming that appears to be like remote procedure call (RPC). The static interface is directly generated in the form of stubs by the IDL pre-compiler. All the methods are specified in advance and are known to the client and the server via client-side stubs and server-side skeletons as proxies, also called surrogates. The particulars of the remote operations are bound to the client code written in high-level language. During the delivery of the invocations to the server, the ORB takes care of all the details involved. Essentially, this client code written to perform static invocations is highly portable across multi-vendor ORBs.

The dynamic method invocation provides a more flexible environment than the static counterpart. Services are only discovered at run time. New methods added to the object implementation do not enforce corresponding changes in the client code. Nevertheless, most applications do not require this level of flexibility and are better off with static stub implementations. The primary advantage of the static invocation is its simplicity, type safety, and efficiency. In fact, it is much easier to program static stubs because remote methods are simply invoked by names and parameters.

7.1 Stubs and Skeletons

The static method invocation is performed through both client-side stubs and server-side skeletons generated by OMG IDL compilers. A stub is a mechanism that effectively creates and issues requests on behalf of a client, while a skeleton is a mechanism that delivers requests to the CORBA object implementation. IDL stubs and skeletons are built directly into the client application and the object implementation, respectively. Not surprisingly, stubs and skeletons are interface-specific because they are directly translated from OMG IDL specifications.

An IDL stub essentially is a stand-in within the local process for the actual target object, so that the client invokes operations of remote server objects just as operations of local objects. The IDL stub is the static invocation interface, representing a language mapping between the client language and the ORB implementation. A set of stub routines is generated from IDL interface specifications and linked into a client program. Each of these stub routines actually corresponds to a particular target object. Subsequently, the client invokes an operation on a remote server object by calling the corresponding stub routine for the target object.

An IDL stub works directly with the client to marshal a request. This involves the conversion from the representation of the request in the programming language to one suitable for transmission over the connection to the target object. On arrival at the server-side ORB, the request is unmarshalled and dispatched to the target object implementation in its programming language form.

7.2 Generating Interface Stubs and Skeletons

Interface stubs and skeletons are generated in the process of creating server classes. Figure 12 illustrates the procedure to prepare server classes for the static method invocations. These following steps are typical of most CORBA implementations [2]:

1. Define object classes in IDL specification, detailing the types of objects, attributes, methods, and parameters. These are the interfaces a server exports to its clients.
2. Run the IDL files through a CORBA-compliant language pre-compiler, producing language skeletons for the server classes.
3. Add the implementation code to the methods in the skeletons.
4. Compile the implementation code through a CORBA-compliant compiler, generating at least three types of output files:
   - import files - describe the objects to an Interface Repository;
   - client stubs for the IDL-defined methods - these stubs are invoked by a client program that needs to statically access IDL-defined services via the ORB;
   - server skeletons - call the methods on the server.

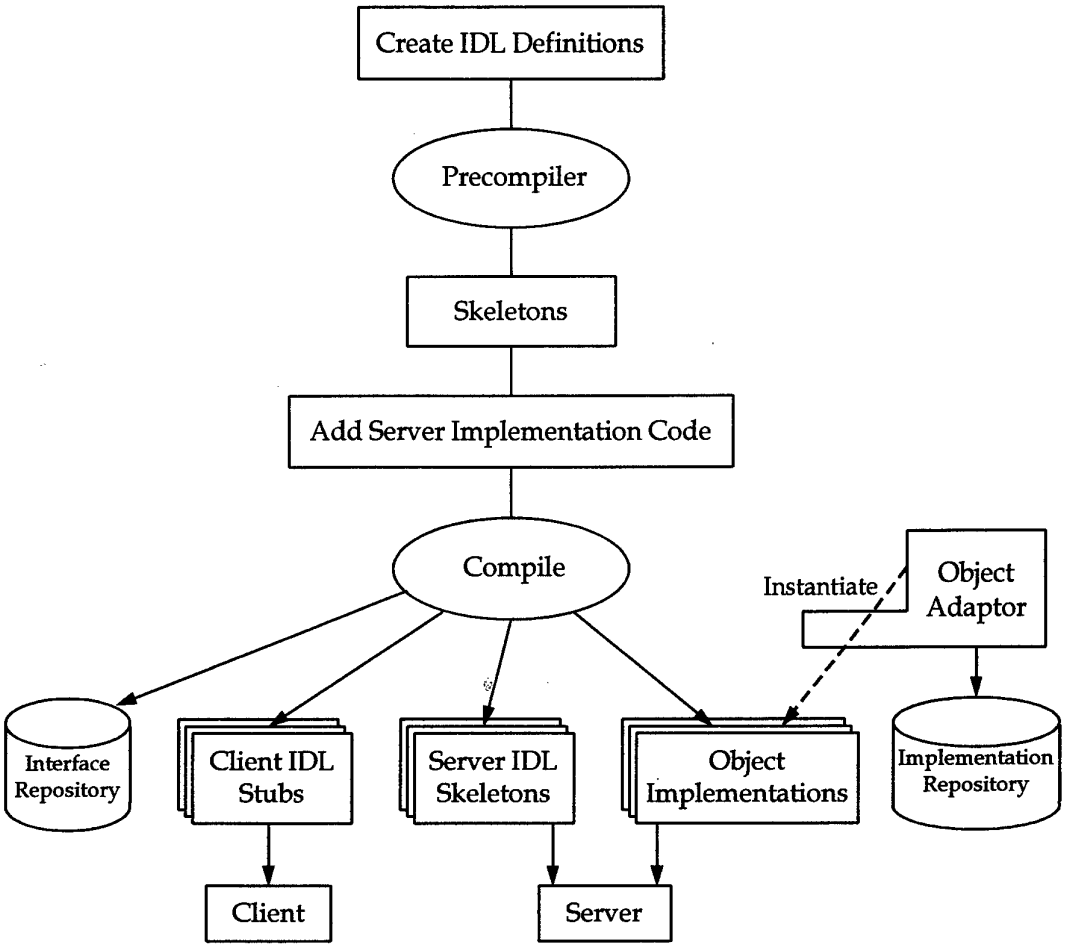


Figure 12 : From IDL to Interface Stubs [2]

5. Bind the class definitions to the Interface Repository, so that programs can access the IDL information at run time.
6. Register the run-time objects with the Implementation Repository, so that object classes supported on a particular server are known. This information is used by the ORB to locate active objects or to request the activation of objects on a particular server. The object adaptor records in the Implementation Repository the object reference and type of any object it instantiates on the server.
7. Instantiate the objects on the server at startup to service remote client method invocations. Run-time objects are instantiated by an object adaptor, as instances of the server application classes.

7.3 Activating Static Invocation

A sequence of events are involved to establish a static invocation between a client and an object implementation across the ORB. The procedure to activate the static method

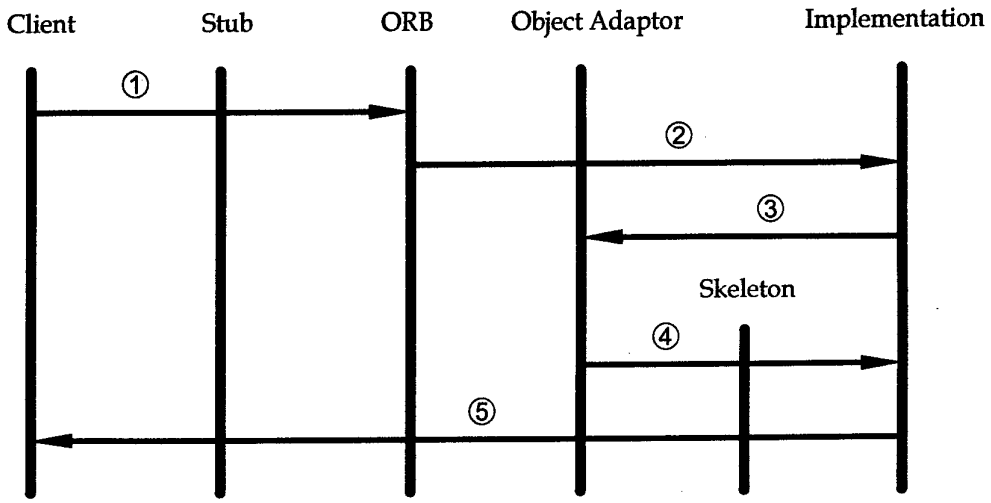


Figure 13 : Activating Static Invocation [5]

invocation is illustrated in Figure 13. The client first calls a remote method through the client stub (Step 1). The ORB delivers the request to the object adaptor, which then activates the object implementation (Step 2). The object adaptor is informed that the object implementation is active and available (Step 3). Upon this confirmation, the object adaptor forwards the request to the implementation via its skeleton (Step 4). Finally, the request is serviced in the object implementation, which returns the result, if any, or an exception in the event of an abnormality, back to the client through the ORB (Step 5).

8. Dynamic Invocation and Dispatch

CORBA supports dynamic invocation so that a client can dynamically invoke the methods of any target object at run time without pre-compiled stubs. A client only discovers interface-related information at invocation time. This dynamic environment considerably facilitates the flexibility and extensibility of remote operations. Hence, servers can offer new services to any client whenever they become available. Similarly, a client is able to use objects whose types or operations might not be known when the client was compiled. Dynamic invocation is useful for interactive programs such as browsers, management support tools and distributed debuggers. These applications can therefore invoke requests on any object without having compile-time knowledge of the object interfaces.

In the CORBA specification, two interfaces are defined to support dynamic invocation: the Dynamic Invocation Interface (DII) for dynamic client request invocation, and the Dynamic Skeleton Interface (DSI) for dynamic dispatch to objects. These two generic interfaces are provided directly by the ORB. They are identical for all ORB

implementations, and thus not dependent on the OMG IDL interfaces of objects being invoked.

8.1 Dynamic Invocation Interface

The Dynamic Invocation Interface (DII) is a generic client-side stub capable of forwarding any request to any object by run-time interpretation of request parameters and operation identifiers. It allows a client application to issue requests for any interface, even if that interface was unknown at the time the application was compiled. Importantly, a server receiving an incoming invocation request does not know whether the client which sent the request used the static or dynamic approach to compose the request.

The dynamic invocation services are part of the CORBA Core[5]. The methods required to prepare for dynamic invocations are collected within four interfaces in the CORBA module. These are CORBA::Object interface, CORBA::Request pseudo-object interface, CORBA::NVList pseudo-object interface, and CORBA::ORB pseudo-object interface. In addition, Interface Repository objects are required to construct a remote invocation.

Pseudo-object interfaces contain serverless object types that do not have object references. These operations are defined in IDL for convenience. The Appendix presents further details of ORB pseudo-object interfaces.

8.1.1 Obtaining an Object Reference

To make dynamic invocations on an object, a client has to find a target object and obtain its reference. This target object reference is used to retrieve the object interface. Subsequently, a request is dynamically constructed and populated with the object reference, the operation name and the parameters.

Remote objects can be discovered by name using the CORBA Naming Service (the CORBA White Pages), or via the CORBA Trader Service (the CORBA Yellow Pages). Once a client discovers the target object, the Dynamic Invocation Interface is able to invoke the operations.

When a reference for a target object is known, a client can obtain the target interface name by invoking the `get_interface()` operation on its reference provided by the CORBA::Object interface. This call returns a reference to an InterfaceDef object located inside an Interface Repository (IR). This reference is an entry point from which the client can further obtain the method description from the IR by issuing the `lookup_name()` and `describe()` operations.

---

[5] The CORBA 2.0 Core contains six components, namely, IDL, ORB interfaces, IR, DII, DSI, and BOA. In fact, this category corresponds to CORBA 1.2 minus the IDL C language mapping, plus modifications resulting from the CORBA 2.0 Requests for Proposal (RFPs) and requirements of Common Object Services Specifications (COSS) 2 Object Services.

8.1.2 Constructing a Request

Parameters in a request are supplied as an NVList pseudo-object that is a list structure of type NamedValue. An empty NVList is created by invoking `create_list()` on a CORBA::ORB object. This parameter list is progressively populated by calling `add_item()` and `add_value()` provided by the CORBA::NVList interface, for each argument in the method. Alternatively, the `create_operation_list()` operation allows the ORB to create an NVList with the parameter descriptions. The client is only required to set each argument value by invoking `add_value()`.

After the argument list is prepared, the client can create a dynamic request for the target object by calling `create_request()` on the object reference with the name of the method, the NVList, and the return value.

To alleviate the laborious task of creating a separate argument list, a client can directly create an empty Request object by calling `_request()` on the object reference with the name of the method. The Request object is populated by progressively invoking `add_arg()` and `add_value()` for each parameter. This technique of creating the Request object is more convenient for invoking methods that do not require parameters.

8.1.3 Invoking the Request

Once a Request pseudo-object has been created and argument values have been embodied within it, the client can submit the request in one of three ways: synchronous invocation, deferred synchronous invocation, or one-way invocation.

1. Synchronous Invocation

   This standard invocation mode is equivalent to an RPC-like synchronous interaction between client and server, in which a request is invoked by calling the `invoke()` operation. The client is however blocked until the response is returned from the object implementation.

2. Deferred Synchronous Invocation

   The request is invoked by calling the `send()` operation which returns control immediately to the client without waiting for the response. This allows the client to continue processing in parallel with the invoked operation. By issuing `get_response()` at a later time, the client can check to determine if a response is available, and if so, the result is collected. Deferred synchronous operations are useful to improve the throughput of a client, particularly in the case of a number of independent long-running invocations.

3. One-Way Invocation

A one-way invocation is specified when the client invokes the request using the `send()` operation with `invoke_flags` set to `INV_NO_RESPONSE`. In this case, no response is expected and there is no way to inform the client of any error.

In addition, the `send_multiple_requests()` operation provided by the CORBA::ORB interface allows deferred synchronous invocations of multiple requests. One-way operations are specified by `INV_NO_RESPONSE` in the `invoke_flags`.

At present, only the DII provides deferred synchronous and one-way request invocations. Table 1 compares the static and dynamic invocations in relation to various communication styles, though this added capability will soon be available to remote requests through static stubs [6]. Nevertheless, the flexibility of the DII mechanism over static stubs comes at a price. Preparing a dynamic request involves defining the target object, method, and parameters through a series of calls to the ORB core services. In particular, the ORB may transparently access the IR to obtain information about the types of the arguments and return value, requiring several remote invocations. In contrast, static invocations do not suffer from the overhead of accessing the IR since the type information has already been compiled into the application. A DII request is therefore less efficient than an equivalent static invocation. A DII request with no arguments and a void return type requires a minimum of two function calls [7]. As a benchmark, comparing the average response times of static versus dynamic Ping invocations shows that a dynamic invocation is about 40 times slower than its static counterpart [2]. In fact, most of the overhead was spent on accessing an IR when a request was prepared.

Table 1 : Invocation Types versus Communication Styles

Communication Styles	*Static Invocation*	*Dynamic Invocation*
Synchronous	Yes	Yes
Deferred Synchronous	No	Yes
One-way	Yes	Yes

8.2 Dynamic Skeleton Interface

The Dynamic Skeleton Interface (DSI) is the server equivalent of the client-side DII, which defines an interface between the ORB and an object implementation through a dynamic skeleton. The DSI enables an ORB to deliver a request to an object implementation that does not have compile-time knowledge of the type of object it is implementing. It provides a run-time binding mechanism for servers that need to handle incoming method calls for components that do not have IDL-based compiled skeletons. The DSI is useful for a certain class of applications, such as interactive software development tools based on interpreters, distributed debuggers and inter-ORB bridges.

Just as the DII allows clients to invoke requests without having access to static stubs, the DSI allows servers to be written without having skeletons for the objects being invoked compiled statically into the program. Usually, an object implementation is connected to the ORB through a static skeleton which is specially built for each object type to match the IDL-defined methods implemented. This static skeleton activates the methods to service requests coming through the ORB and the object adaptor. A dynamic skeleton can replace a static skeleton for any type by serving the same architectural role as a type-specific, IDL-based skeleton. Specifically, a dynamic skeleton examines parameter values in incoming requests to determine the target object and method.

The DSI receives detailed specifications of the operation to be invoked as arguments. Without prior knowledge of the object type, all incoming requests through the DSI are prepared using the Dynamic Implementation Routine (DIR) before connecting to an object implementation. The arguments that are delivered to a DIR consist of an arbitrary object reference and a request packaged as a ServerRequest pseudo-object. The ServerRequest pseudo-object is analogous to the Request pseudo-object in the DII.

9. The Interface Repository

The Interface Repository (IR) is a run-time database that contains the interface specifications of each object an ORB recognises. It provides for the storage, distribution, and management of a collection of related interface definitions specified in OMG IDL. The ORB can use these object definitions to interpret and handle the values provided in a request, such as type checking of method signatures, and checking of interface inheritance graphs. Also, the IR is a convenient place in the ORB for developers to stash additional interface information such as debugging information, stub and skeleton libraries.

A CORBA object becomes self-describing after its interface information is installed in the IR. An ORB may access multiple IRs. Conversely, an IR may be accessed by multiple ORBs. In particular, a multi-ORB federation of interface repositories enables objects to go across heterogeneous ORBs.

9.1 The Containment Hierarchy of Interface Repository Classes

Interface definitions are represented as sets of objects that contain descriptions for the operations, exceptions, context objects, and parameter types. CORBA metadata information is grouped into modules that represent naming spaces. The IR uses modules as a way to navigate through those groups by name using a hierarchical traversal approach.

In the IR, each interface is represented by an interface object. Although IR operations and the conceptual framework are object-oriented, a non-object-based implementation of the IR can also provide an object-oriented interface to the users. An interface object maintained in the IR can be one of the following IDL structures:

- Repository: a top-level organisational object as the root for all the modules contained in a repository namespace.
- ModuleDef: a logical grouping of objects representing interfaces, types, constants, exceptions, and other ModuleDef objects.
- InterfaceDef: an object interface containing attributes, constants, types, operations, and exceptions.
- OperationDef: an operation or method on an object interface, containing parameters and exceptions raised by this operation.
- ParameterDef: an object representing a parameter of an operation.
- AttributeDef: an attribute of an interface.
- ConstantDef: an object representing a named constant.
- ExceptionDef: an exception that can be raised by an operation.
- TypeDef: a type definition as part of an IDL definition.

Figure 14 illustrates the possible containment relationships between these object types. In particular, instances of the Repository contain other objects. Instances of the ModuleDef, InterfaceDef, and OperationDef classes are both contained and containers, whereas instances of the ConstantDef, TypeDef, ParameterDef, and ExceptionDef classes are always contained in other objects.

In the Interface Repository class hierarchy, three abstract superclasses are defined: IRObject, Contained, and Container. All IR objects inherit from the IRObject interface that provides an operation for identifying the actual type of the object. Objects that are containers inherit navigation operations from the Container interface. These are the Repository, ModuleDef, and InterfaceDef classes. The Contained interface defines the behaviour of objects contained in other objects, which is inherited by all the IR objects except the Repository class.

Besides supporting dynamic invocation, the IR can also be used as a source for generating static support code for CORBA-based applications. Both the static and dynamic invocation interfaces can use information stored in the IR to facilitate type checking of objects at run time.

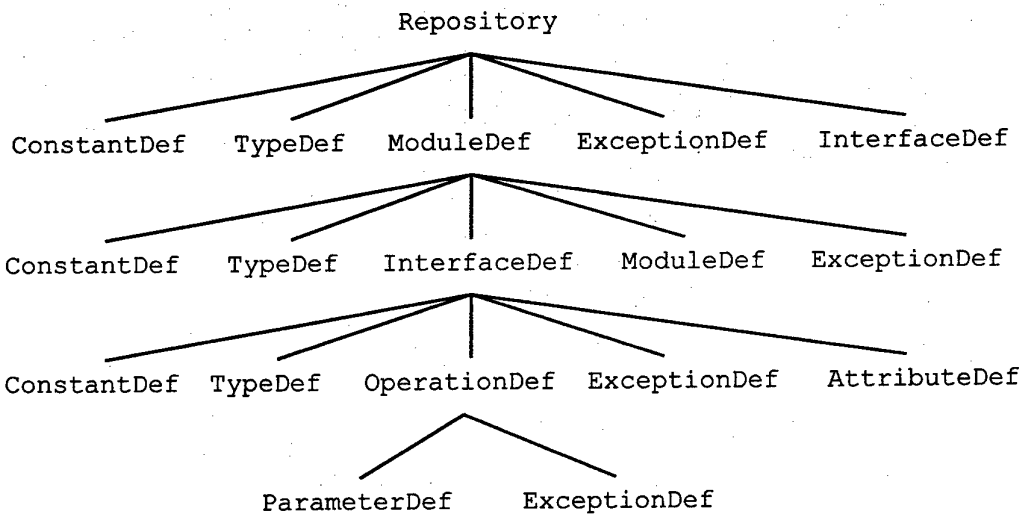


Figure 14 : Containment Hierarchy for the Interface Repository Classes [2]

9.2 Interface Retrieval

Using an Interface Repository, a client is able to locate an object unknown at compile time, and inquire about its interface. Based on this information, the client can construct a request to the selected target object through the Dynamic Invocation Interface. There are three ways to retrieve interface information from the Interface Repository:

1. An InterfaceDef object can be retrieved from any valid object reference using the `get_interface()` operation defined in the CORBA::Object interface. Since all interfaces are derived from CORBA::Object, every object therefore supports the `get_interface()` operation. The target interface is fully described in the InterfaceDef object.
2. An interface can be located by navigating through the module namespace if the name of the interface is known. Starting at the root module of the repository, the desired module is located by traversing over all of the module definitions. The target module is then opened and iterated in the same manner over all the definitions. When the entry is found, invoking the InterfaceDef::`describe_interface()` operation returns the metadata that describes the interface.
3. A Repository object can be used to look up any definition either by name or identifier. Given a particular Repository ID, an InterfaceDef object can be located by invoking the Repository::`lookup_id()` operation. The metadata about the interface can be obtained from this InterfaceDef object.

9.3 Federated Interface Repositories

It is possible to create federations of interface repositories operating across multiple ORBs. To avoid name clashes and to synchronise definitions across ORBs and repositories, unique IDs, called `Repositoryids`, are assigned to global modules, interfaces, constants, typedefs, exceptions, attributes, operations, and parameters. Using these global repository IDs preserve the identity of an interface across ORB and repository boundaries.

The format of the `Repositoryid` is a short format name followed by a colon (":") followed by characters according to the format. Three formats are defined in CORBA 2.0 specification: the OMG IDL format, the DCE Universal Unique Identifier (UUID) format, and a local format.

9.3.1 OMG IDL Format

IDL scoped names are used in the OMG IDL format for `Repositoryids`. An optional unique prefix, major and minor version numbers are also included. This format consists of three components separated by colons ":". The first component is the format name, "IDL". The second component is a list of identifiers separated by "/" characters. The first identifier on the list is a unique prefix, and the rest are the IDL identifiers that make up a scoped name. The third component consists of major and minor versions in decimal format separated by a period ".".

For example, a valid `Repositoryid` for the interface `Cat` in the module `MyAnimals` shown in Figure 11 is `"IDL:DogCatInc/MyAnimals/Cat/:1.0"`. An organisation name, `DogCatInc`, is taken as a unique prefix in this case [2].

9.3.2 DCE Universal Unique Identifier (UUID) Format

The DCE UUID format for `Repositoryids` consists of three components separated by colons ":". The first component is the format name "DCE", followed by the printable form of the UUID as the second component. The third component is a minor version number in decimal format. A UUID is a globally unique number in DCE generated using the current date and time, a network card ID, and a high-frequency counter. An example of a DCE `Repositoryid` is `"DCE:700dc518-0110-11ce-ac8f-0800090b5d3e:1"`.

9.3.3 Local Format

Only for use in an isolated repository, the local format for `Repositoryids` consists of the format name "LOCAL", a colon ":", followed by an arbitrary string. This format is intended for short-term use such as in a development environment. A simple example to generate unique local IDs is making use of consecutive integers.

10. The ORB Interface

The ORB Interface is the interface to the ORB functions to be accessed directly by both the client-side and the implementation-side architecture. These operations are the same for all ORBs and thus must be supported by any ORB implementation.

The operations are implemented in the ORB-based environment. Some of these operations appear to be on the ORB, others appear to be on an object reference. These ORB-implemented operations are described as operations on objects for model consistency. The concept of modelling all functions as being performed by objects allows all ORB functions to be defined as operations in IDL interfaces.

The ORB interface also defines additional operations for creating lists and retrieving the default context used in the Dynamic Invocation Interface.

10.1 Converting Object References to Strings

An object reference is opaque and ORB-dependent, which is not, however, a convenient characteristic for persistent storage. Besides, object references cannot be passed from one application to another. To facilitate storage and communication of object references, the ORB Interface provides the operation `object_to_string()` for converting object references to strings that a client can store or transmit. The subsequent operation `string_to_object()` returns the original object reference.

10.2 Object Reference Operations

The ORB Interface supports additional operations that can be applied to any object reference. These operations on object references are directly implemented by the ORB, not by the object implementation that the object reference refers to. These operations are defined as part of the Object interface.

The `create_request()` operation creates a Request pseudo-object for an object. The `get_interface()` operation returns an object in the Interface Repository associated with the object, which can be further used to extract meta-information regarding the object type. The `get_implementation()` operation returns an object in the Implementation Repository that describes the implementation of the object.

The `duplicate()` operation creates an additional object reference to a particular object. The `release()` operation reclaims the storage of an object reference which is no longer used by a program. The `is_nil()` operation tests an object reference for referencing no object, while the `is_a()` operation determines if an object is really an instance of a shared type identifier. The `non_existent()` operation tests whether an object has been destroyed.

To efficiently manage object reference identity, two identity-related operations are provided. The `hash()` operation produces an internal hash value for an object reference. Two object references are not identical if they hash differently. The `is_equivalent()` operation determines if two object references are equivalent. Two object references are equivalent if they are identical or they refer to the same object.

11. The Basic Object Adaptor

An object adaptor is the main interface through which object implementations access most ORB services and the ORB core. It isolates object implementations from the ORB core using three kinds of interfaces: an ORB-specific interface to the ORB core, an ORB-specific interface to the implementation skeletons, and a public interface to the object implementations. An object adaptor acts on behalf of the server objects to monitor the ORB core communication services, and to accept requests for service. It is also a service layer between the implementation skeletons and the ORB core. An object adaptor is primarily responsible for activating and deactivating objects and implementations. In addition, it provides the run-time environment for generating object references, method invocation, implementation registration, instantiating server objects, and request authentication.

A number of object adaptors are required to support a wide variety of object implementations with differing characteristics such as granularities, lifetimes, policies, and usage. The Basic Object Adaptor (BOA) is a general object adaptor fully defined in the CORBA specification to support a wide range of CORBA-compliant object implementations. Object adaptors can be tailored to provide the necessary functionality for very specialised requirements. Examples of special-purpose object adaptors are the library object adaptor for objects that have library implementations, and the object-oriented database adaptor to provide access to objects stored in an object-oriented database.

The Basic Object Adaptor (BOA) is packaged as the BOA interface, which provides the following functions:

- generation and interpretation of object references;
- authentication of the principal making the call;
- activation and deactivation of the implementation;
- activation and deactivation of individual objects; and
- method invocation through skeletons.

An object implementation is registered in an Implementation Repository, which also maintains platform-specific information describing the object implementation as ImplementationDef objects. The BOA uses this information to start up the object server.

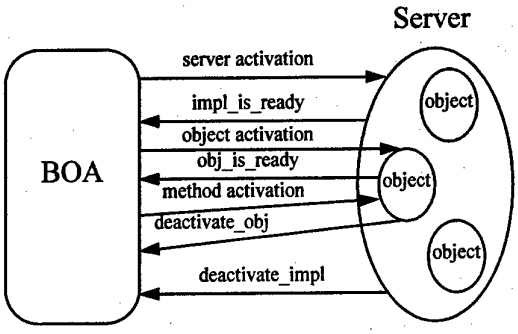


Figure 15 : Shared Server Activation Policy [4]

The Implementation Repository also contains additional information for debugging and administration. The BOA makes use of both the Interface Repository and the Implementation Repository to associate object references with their interfaces and implementations.

There are two kinds of activation that the BOA needs to perform as part of operation invocation: implementation activation and object activation. Implementation activation occurs when no implementation for a particular object is currently available to handle the request. The BOA starts up the implementation using operating system facilities. After the implementation initiates itself and responds with the BOA operation `impl_is_ready()` or `obj_is_ready()` for per-object servers[6], requests are delivered to the implementation methods through the skeletons. Object activation occurs when no instance of the object is available to handle the request, though the implementation has already been activated.

According to the roles and interactions of the implementations, the objects, and the BOA, the CORBA specification defines four policies that all BOA implementations support for implementation activation:

1. Shared Server Activation Policy

   In a shared server activation policy as illustrated in Figure 15, multiple active objects of a given implementation share the same server. The BOA activates the server the first time a request is invoked on any object implemented by that server. Once initialised, the server notifies the BOA by calling `impl_is_ready()`. The BOA delivers all subsequent requests to this server process. The BOA does not activate another server process for that implementation. Upon receipt of the first request for a particular object implemented by that server, the BOA calls the object activate routine to ensure that the object is ready to service requests. When the server is ready to terminate, it notifies the BOA by calling `deactivate_impl()`. The server process can deactivate a particular object at any time by issuing

[6] The CORBA specification defines the term server as a separately executable entity or process that the BOA can start on a particular system. An object implements an interface, while a server can contain one or more objects.

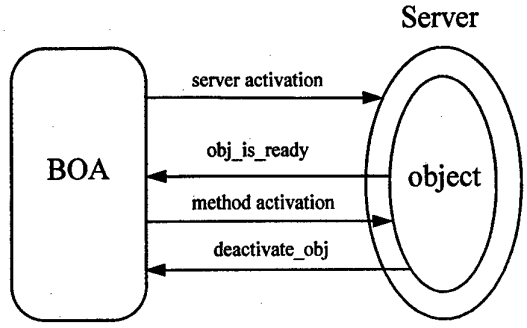


Figure 16 : Unshared Server Activation Policy [4]

`deactivate_obj()`. Most CORBA servers operate under this policy by using threads to run multiple objects concurrently within the same process [2].

2. Unshared Server Activation Policy

   Figure 16 shows an unshared server activation policy in which only one object of a given implementation can be active at any one time in one server. A new server is activated the first time a request is invoked on the object. After the server initiates itself, the BOA is notified using the `obj_is_ready()` operation. The server informs the BOA of its termination by calling `deactivate_impl()`. An unshared server is applicable to situations where a dedicated object is required, such as a printer or a robot on a manufacturing line.

3. Server-per-Method Activation Policy

   A server-per-method activation policy is shown in Figure 17. Each invocation of a method is implemented by a separate server being started, with the server terminating when the method completes. Several servers for the same object or even the same method of the same object may be active simultaneously. As each request starts a new server, the BOA is not informed whether the implementation is ready or deactivated. This activation policy is useful for running scripts or utility programs that execute once and then terminate [2].

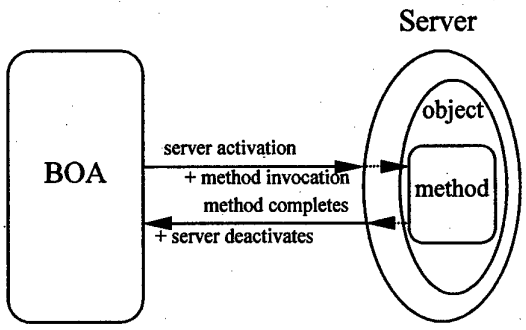


Figure 17 : Server-per-Method Activation Policy [4]

4. Persistent Server Activation Policy

   Under a persistent server activation policy, the server is activated by some other means outside the BOA. The server still registers with the BOA using the `impl_is_ready()` operation. The BOA treats the server as a shared server. It sends activations for individual objects and method calls to a single process. If no implementation is ready when a request arrives, an error is returned for that request. A persistent server is just a special case of a shared server in which the server activation is not performed by the ORB. DBMS, TP monitor, and web server are expected to be good candidate applications for a persistent server activation policy [2].

12. Inter-ORB Architecture

Direct ORB-to-ORB interoperability is possible when two ORBs in the same domain[7] understand the same object references and IDL type structure. In any case, the success of the CORBA approach to distributed object computing depends greatly on the ease with which objects across different ORB implementations can call on one another's services. Hence, the CORBA 2.0 specification defines a general ORB interoperability architecture to support distributed objects across multiple heterogeneous ORBs.

ORBs from separate domains can only communicate through a bridge that maps ORB-specific information from one ORB domain to the other, so that users of any ORB only see their appropriate content and semantics. Essentially, this mapping mechanism is achieved via two approaches: immediate bridging and mediated bridging. Immediate bridging, or a full bridge, provides one-to-one protocol translation. This is a simple and effective solution as long as the number of protocols remains small. Since it is not practical to provide all possible kinds of full bridges among an increasing number of ORB protocols, mediated bridging, or a half bridge, creates a common backbone protocol between different ORB domains.

The CORBA 2.0 specification is designed to reduce the number of different combinations of half bridges between ORB domains. The general ORB interoperability architecture is based on the General Inter-ORB Protocol (GIOP), which specifies a set of message formats and common data representations for ORB-to-ORB interactions. The GIOP is a very basic inter-ORB protocol that serves as a common backbone over any connection-oriented transport protocol. The GIOP defines all the ORB request and reply semantics, and makes use of the common data representation for mapping IDL data types into a flat, networked message representation. The GIOP is designed to be simple and easy to implement while still allowing for reasonable scalability and performance.

[7] A domain is a distinct scope within which certain common characteristics are exhibited and common rules are observed.

The Internet Inter-ORB Protocol (IIOP) specifies how the GIOP is built over a TCP/IP network. The Internet is used as a backbone ORB through which other ORBs can bridge. The IIOP is basically TCP/IP with some CORBA-defined message exchanges for out-of-the-box interoperability among TCP/IP based ORBs. For an ORB complying with the CORBA 2.0 specification, support for the GIOP and the IIOP is mandatory. Such an ORB must either implement IIOP natively or provide a half-bridge that translates requests to and from the global IIOP backbone.

The ORB interoperability architecture also defines Environment-Specific Inter-ORB Protocols (ESIOPs) to allow ORBs to be built for special situations. ESIOPs are optimised for particular environments such as the Distributed Computing Environment (DCE). The DCE Common Inter-ORB Protocol (DCE-CIOP) is an ESIOP, to be used by ORBs in environments where DCE is already installed. This allows for easier integration of CORBA and DCE applications. Support for the DCE-CIOP or any other ESIOP by a CORBA 2.0 ORB is optional.

13. Concluding Remarks

Use of the OMG Object Management Architecture (OMA) is now emerging as a key strategy for supporting interoperable applications based on distributed interoperable objects. Adoption of this approach is motivated by a desire to develop software with reusable components that interact through well-defined interfaces. As the communications heart of this architectural framework, CORBA provides a flexible communication and activation substrate for distributed heterogeneous object computing environments.

CORBA provides an object-oriented approach to integrating legacy applications through well-defined interfaces. Any implementation, including non-object-oriented implementations and legacy software, can be made to provide object-oriented interfaces. Encapsulation of these systems becomes feasible by means of object-oriented wrappers or adaptors.

The language independence feature of OMG Interface Definition Language (IDL) facilitates isolation of interface definitions from object implementations. It allows objects to be constructed using different programming languages, and yet, to still communicate with one another. Language-independent interfaces are important within heterogeneous systems, since not all programming languages are supported or available on all platforms. Given the inevitable heterogeneity of distributed object systems, the simplicity of OMG IDL is critical to the success of CORBA as an integration technology. For communication between ORBs, IIOP provides a common way to connect distributed objects across the Internet and intranets. Consequently, CORBA is becoming almost as ubiquitous as TCP/IP, creating a mass market for

components that run on top of CORBA middleware. For example, Visigenic ORB is now incorporated in all Netscape browsers and servers, which become IIOP-compliant.

After the creation of the CORBA specification, the OMG is shifting the power to set its own technical directions, resulting in a change of OMG focus from the CORBA component to other higher level components of the OMA. A collection of OMG Object Services, called CORBAservices, is provided for construction of higher level facilities and object frameworks. These include Naming, Events, Life Cycle, Persistence, Relationships, Externalisation, Transactions, Concurrency Control, Licensing, Query, Properties, Security, Time, Collections, and Trading Services. These are the basic building blocks for a distributed object infrastructure.

The CORBA specification has laid the groundwork for interoperability, and now domain-specific business processes and requirements need to be addressed. Higher level services are collectively called OMG Common Facilities, or CORBAfacilities, providing standardised interfaces to common application services that are applicable to most domains. Both CORBAservices and CORBAfacilities are the upper layers that can be applied differently in various vertical market segments such as financial systems or CAD systems. As a consequence of these efforts, the development of standard OMG specifications is moving towards the realisation of a true commercial off-the-shelf (COTS) software component marketplace. The OMG will continue working to help create a market in which buying and using software components in distributed heterogeneous environments becomes a reality.

In addition, the OMG has set up a number of task forces and special interest groups, which cover nearly the entire spectrum of topics related to distributed computing. Of particular interest to defence is the Command, Control, Computing, Communications and Intelligence Special Interest Group (C4I SIG) whose goal is to define a CORBA-based framework for C4I facilities. Examples of the focus areas have included real-time CORBA, secure CORBA, CORBA C4I common messaging, and CORBA C4I business objects for more effective C4I systems implementation.

It is not uncommon that software components of defence applications reside on a number of diverse computers and operation systems. The CORBA as a middleware technology can simplify the construction of these applications by providing standardised mechanisms that distributed components can use to communicate over a network. Furthermore, the broad availability of the CORBA standard supports a wide range of platforms and programming languages. Its support for object-oriented software is an especially good choice for integrating diverse types of systems. Another important factor is its accompanying long-term OMA for defining and integrating supporting services. Importantly, applying CORBA technology to C4I problems in the military environment provides simple integration of legacy software and COTS software.

References

[1] John R. Nicol, C. Thomas Wilkes, and Frank A. Manola, "Object Orientation in Heterogeneous Distributed Computing Systems," IEEE Computer, June 1993, pp. 57-67.

[2] Robert Orfali, Dan Harkey, and Jeri Edwards, "Instant CORBA," John Wiley & Sons, 1997.

[3] "A Discussion of the Object Management Architecture," Object Management Group, January 1997.

[4] Ron Ben-Natan, "CORBA: A Guide to the Common Object Request Broker Architecture," McGraw-Hill, 1995.

[5] Alan Pope, "The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture," Addison-Wesley, 1997.

[6] Steve Vinoski, "CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments," IEEE Communications, February 1997, pp. 46-55.

[7] Steve Vinoski, "Distributed Object Computing with CORBA," C++ Report, vol. 5, July/August 1993.

Appendix – ORB Pseudo-Objects

ORB and object adaptor functionalities are usually presented as pseudo-objects, which have interfaces but may not actually be implemented by an object. A pseudo-object has the appearance of an object implementation with an IDL-defined interface just like any other, but the implementation may simply trap these operations from within the ORB and service them in the ORB. The concept of modelling all functions as being performed by objects, even when this is not the case, is useful. It permits all functions to be defined as operations in IDL interfaces.

A pseudo-object cannot be invoked with the dynamic interface because it does not have object references. A pseudo-object does not inherit from CORBA::Object, the root interface of all CORBA objects. Apart from the standard OMG IDL types (see Figure 3) which must be available in all ORB implementations, the ORB pseudo-objects should also be made available in any language mapping.

A.1 Environment Interface

The Environment interface provides a vehicle for dealing with exceptions in those cases where true exception mechanisms are unavailable or undesirable (for example in the DII). They may be set and inspected using the `exception` attribute. The `clear()` function causes the Environment to delete any Exception it is holding.

```
interface Environment
{
   attribute exception exception;
   void clear();
};
```

A.2 Request Interface

The Request interface provides the primary support for DII. Both the Object and Request interfaces contain operations that permit a client to dynamically create and invoke a request for an object of arbitrary type to perform an arbitrary operation. A new request on a particular target object may be constructed using the short version, or one of the long forms of the request creation operation in the Object interface. Alternatively, arguments and contexts may be added after construction via the corresponding attributes in the Request interface. The Request interface also allows the client application to supply all information necessary for the invocation without requiring the ORB to utilise the Interface Repository.

The Request interface is used to instantiate CORBA request objects to perform operations. The `add_arg()` operation incrementally adds arguments to the request. An appropriate method is invoked using the `invoke()` operation. The `send()` function initiates a deferred synchronous operation according to the information in the

Request. It returns control to the caller without waiting for the operation to finish. The `delete()` operation deletes the request and reclaims all resources taken up by the request once it has been completed. The `get_response()` operation determines whether the request has completed. The `RESP_NO_WAIT` response flag indicates that the caller does not want to wait for a response even if the request is still in progress.

```
interface Request
{
   Status add_arg(
           in Identifier name,   // argument name
           in TypeCode arg_type, // argument datatype
           in void * value,      // argument value to be added
           in long len,     // length/count of argument value
           in Flags arg_flags    // argument flags
   );
   Status invoke(
           in Flags invoke_flags // invocation flags
   );
   Status delete();
   Status send(
           in Flags invoke_flags // invocation flags
   );
   Status get_response(
           in Flags response_flags    // response flags
   );
};
```

A.3 Context Interface

The Context interface supplies optional context information associated with a method invocation. A Context object contains a list of properties, each consisting of a name and an associated string value. Context properties represent information about the client, environment, or circumstances of a request that are inconvenient to pass as parameters.

The `set_one_value()` operation sets a single context object property, whereas the `set_values()` operation sets one or more property values in the context object. The `get_values()` operation retrieves the specified context property value(s). This operation can use a wildcard to match all properties with a name matching the specified part and anything in the segments which is a wildcard. The `delete_values()` operation deletes the specified property value(s) from the context object. The `create_child()` operation creates a child context object. The indicated context object is deleted using the `delete()` operation.

```
interface Context
{
   Status set_one_value(
```

```
        in Identifier prop_name,    // property name to add
        in string value             // property value to add
    );
    Status set_values(
        in NVList values      // property values to be changed
    );
    Status get_values(
        in Identifier start_scope, // search scope
        in Flags op_flags,         // operation flags
        in Identifier prop_name,// name of property(s) to retrieve
        out NVList values          // requested property(s)
    );
    Status delete_values(
        in Identifier prop_name // name of property(s) to delete
    );
    Status create_child(
        in Identifier ctx_name,     // name of context object
        out Context child_ctx       // newly created context object
    );
    Status delete(
        in Flags del_flags    // flags controlling deletion
    );
};
```

A.4 ORB Interface

The ORB interface is the programmer interface to the Object Request Broker. Object references may be translated into string form by the `object_to_string()` operation to facilitate storage or communication of object references. Subsequently, the `string_to_object()` operation returns the corresponding object reference.

The `create_list()` operation creates a pseudo-object by allocating a list structure of the specified size for initial use. The `create_operation_list()` operation returns an NVList pseudo-object initialised with the argument descriptions that may be used in dynamic invocation requests. The `create_named_value()` operation creates NamedValue objects to be used as return value parameters for the `create_request()` operation provided by the Object interface. When a request is invoked, the `create_exception_list()` and `create_context_list()` operations create an ExceptionList providing a list of TypeCodes for all user-defined exceptions, and a ContextList providing a list of Context strings, respectively. A reference to the default process Context pseudo-object is returned using the `get_default_context()` operation. The `create_environment()` operation constructs an Environment.

The `send_multiple_requests()` operation initiates more than one request in parallel. If the `INV_NO_RESPONSE` invocation flag is set, it is an one-way operation and the invoker does not intend to wait for a response. The `get_next_response()`

operation returns the next request that completes. The RESP_NO_WAIT response flag indicates that the caller does not want to wait for a response even if there are no completed requests pending.

Specific to object initialisation, the BOA_init() operation obtains a reference to the BOA pseudo-object for object registration with the ORB. Invoking list_initial_services() returns a list of names to well-known services, whereas the resolve_initial_references() operation converts these string names of services to object references.

```
interface ORB
{
   string object_to_string(in Object obj);
   Object string_to_object(in string str);
   Status create_list(
           in long count,
           out NVList new_list
   );
   Status create_operation_list(
           in OperationDef oper,
           out NVList new_list
   );
   Status create_named_value(out NamedValue nmval);
   Status create_exception_list(out ExceptionList exclist);
   Status create_context_list(out ContextList ctxlist);

   Status get_default_context(out Context ctx);
   Status create_environment(out Environment new_env);

   Status send_multiple_requests(
                in RequestSeq req,
                in Flags invoke_flags
   );
   Status get_next_response(
                out Request req,
                in Flags response_flags
   );

   BOA BOA_init(
           inout arg_list argv,
           in OAid boa_identifier
   );
   ObjectIdList list_initial_services();
   Object resolve_initial_references(in ObjectId identifier)
      raises(InvalidName);
};
```

A.5 BOA Interface

The BOA interface mediates between the ORB and the object implementation by providing operations that the object implementation can access. Once it initialises itself, the implementation notifies the BOA that it is prepared to handle requests by calling `impl_is_ready()` or `obj_is_ready()` for per-object servers. The server remains active and will receive requests until it calls `deactivate_impl()`. The `deactivate_obj()` operation is used to deactivate objects that run within implementations. The `create()` operation is used to describe the implementation of a new object instance to the BOA and obtain a reference. The reference data associated with an object is obtained using the `get_id()` operation. The `change_implementation()` operation updates the implementation information associated with an existing object. An object reference is destroyed by the `dispose()` operation. This handles the destruction of the object only as far as the BOA and the ORB are concerned, the actual object must be destroyed and resources deallocated by the implementation. The object implementation can obtain the principal on whose behalf the request is performed by the `get_principal()` operation.

```
interface BOA
{
   Object create(
           in ReferenceData id,
           in InterfaceDef intf,
           in ImplementationDef impl
   );
   void dispose(in Object obj);
   ReferenceData get_id(in Object obj);
   void change_implementation(
                in Object obj,
                in ImplementationDef impl
   );
   Principal get_principal(
                in Object obj,
                in Environment ev
   );
   void set_exception (
      in exception_type major,    // NO, USER, or
                                  // SYSTEM_EXCEPTION
      in string userid,           // exception type id
      in void *param              // pointer to associated data
   );
   void impl_is_ready(in ImplementationDef impl);
   void deactivate_impl(in ImplementationDef impl);
   void obj_is_ready(
           in Object obj,
           in ImplementationDef impl
   );
   void deactivate_obj(in Object obj);
```

```
};
```

A.6 TypeCode Interface

In CORBA, TypeCodes are defined to represent each of the IDL-defined data types. A TypeCode is an association of a kind with a parameter list. It is used as a coding object for types which are part of many IDL declarations. These TypeCodes create self-describing data that can be passed across operating systems, ORBs, and Interface Repositories. The TypeCode interface defines a set of methods that operate on TypeCodes.

The `BadKind{}` exception is raised if an operation that is not appropriate for the TypeCode kind is invoked. The `Bounds{}` exception is raised by an operation if the index parameter is greater than or equal to the number of members constituting the type.

Applying the `equal()` operation to equal TypeCodes gives identical results. The `kind()` operation determines what other operations can be invoked on the TypeCode. The `id()` operation returns the RepositoryId globally identifying the type. The `name()` operation gets the simple name identifying the type within its enclosing scope. Invoking `member_count()` on structure, union, and enumeration returns the number of members constituting the type, while `member_name()` returns the simple name of the member identified by `index`. The `member_type()` operation invoked on structure and union obtains the TypeCode describing the type of the member identified by `index`. Invoking only on union, `member_label()` returns the label of the union member identified by `index`, `discriminator_type()` returns the type of all non-default member labels, and `default_index()` returns the index of the default member. The `length()` operation gets the bound for strings and sequences, or the number of elements for arrays. The `content_type()` operation gives the element type for sequences and arrays, or the original type for aliases. The `param_count()` and `parameter()` operations only provide access to those parameters that were present in previous versions of CORBA.

```
interface TypeCode
{
   exception Bounds{};
   exception BadKind{};

   boolean equal(in TypeCode tc);
   TCKind kind();

   RepositoryId id() raises(BadKind);
   Identifier name() raises(BadKind);

   unsigned long member_count() raises(BadKind);
   Identifier member_name(in unsigned long index)
```

```
      raises(BadKind, Bounds);

   TypeCode member_type(in unsigned long index) raises(BadKind,
      Bounds);

   any member_label(in unsigned long index) raises(BadKind,
      Bounds);
   TypeCode discriminator_type() raises(BadKind);
   long default_index() raises(BadKind);

   unsigned long length() raises(BadKind);

   TypeCode content_type() raises(BadKind);

   long param_count();
   any parameter(in long index) raises(Bounds);
};
```

A.7 Principal Interface

The Principal interface represents information about principals requesting operations. There are no defined operations.

```
interface Principal{};
```

A.8 NVList Interface

The NVList interface defines operations to construct parameter lists. An NVLIst object maintains a list of self-describing data items called NamedValues. The `add()` function creates an unnamed value, initialising only the flags. The `add_item()` function adds a new item to the indicated list, whereas the `add_value()` function initialises name, value, and flags. The total number of items in the list is returned by the `get_count()` function. The `item()` function can be used to access existing elements. The `free()` operation frees the list structure and any associated memory, while the `free_memory()` operation frees any dynamically allocated out-arg memory associated with the list. An item can be removed from the list by invoking `remove()`.

```
interface NVList
{
   NamedValue add(in Flags flags);
   NamedValue add_item(
                in Identifier item_name,    // name of item
                in Flags flags              // item flags
   );
   NamedValue add_value(
                     in Identifier item_name,
                     in any val,            // item value
                     in Flags flags
```

```
    );
    Status get_count(
            out long count   // number of entries in the list
    );
    NamedValue item(in unsigned long index) raises(Bounds);
    Status free();
    Status free_memory();
    Status remove(in unsigned long index) raises(Bounds);
};
```

DISTRIBUTION LIST

A Primer of CORBA: A Framework for Distributed Applications in Defence

T. A. Au

AUSTRALIA

DEFENCE ORGANISATION

C3ID Branch
DGC3ID
DCD
DOIS

S&T Program
Chief Defence Scientist
FAS Science Policy
AS Science Corporate Management
} shared copy
Director General Science Policy Development
Counsellor Defence Science, London (Doc Data Sheet)
Counsellor Defence Science, Washington (Doc Data Sheet)
Scientific Adviser to MRDC Thailand (Doc Data Sheet)
Scientific Adviser Policy and Command
Navy Scientific Adviser (Doc Data Sheet and distribution list only)
Scientific Adviser - Army (Doc Data Sheet and distribution list only)
Air Force Scientific Adviser (Doc Data Sheet and distribution list only)

Aeronautical and Maritime Research Laboratory
Director

Electronics and Surveillance Research Laboratory
Director

Chief of Communications Division
Research Leader Military Information Networks
Head Network Integration Group
Author: T. A. Au

DSTO Library
Library Fishermens Bend
Library Maribyrnong
Library Salisbury (2 copies)
Australian Archives
Library, MOD, Pyrmont (Doc Data sheet only)
*US Defense Technical Information Center, 2 copies
*UK Defence Research Information Centre, 2 copies
*Canada Defence Scientific Information Service, 1 copy
*NZ Defence Information Centre, 1 copy
National Library of Australia, 1 copy

Capability Development Division

Director General Maritime Development (Doc Data Sheet only)
Director General Land Development (Doc Data Sheet only)
Director General Aerospace Development (Doc Data Sheet only)

Navy

Communications School (Navy), HMAS Cerberus, Hastings, Victoria.

Army

SO (Science), DJFHQ(L), MILPO Enoggera, Queensland 4051 (Doc Data Sheet only)
NAPOC QWG Engineer NBCD c/- DENGRS-A, HQ Engineer Centre Liverpool Military Area, NSW 2174 (Doc Data Sheet only)
School of Signals, Simpson Barracks, Macleod, Victoria.

Air Force

School of Technical Training, RAAF Base, Wagga, NSW 2651.

Acquisitions Program

PD JCSE
PD JISE
PD BCSS

Intelligence Program

DGSTA Defence Intelligence Organisation

Corporate Support Program

OIC TRS, Defence Regional Library, Canberra

SPARES (5 copies)

Total number of copies: **49**

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION
DOCUMENT CONTROL DATA

1. PRIVACY MARKING/CAVEAT (OF DOCUMENT)

2. TITLE

A Primer of CORBA: A Framework for Distributed Applications in Defence

3. SECURITY CLASSIFICATION (FOR UNCLASSIFIED REPORTS THAT ARE LIMITED RELEASE USE (L) NEXT TO DOCUMENT CLASSIFICATION)

Document	(U)
Title	(U)
Abstract	(U)

4. AUTHOR(S)

T. A. Au

5. CORPORATE AUTHOR

Electronics and Surveillance Research Laboratory
PO Box 1500
Salisbury SA 5108 Australia

6a. DSTO NUMBER	6b. AR NUMBER	6c. TYPE OF REPORT	7. DOCUMENT DATE
DSTO-GD-0192	AR-010-622	General Document	March 1999

8. FILE NUMBER	9. TASK NUMBER	10. TASK SPONSOR	11. NO. OF PAGES	12. NO. OF REFERENCES
E8709/4/19 (1)	ADF96/295	DGC3ID	48	7

13. DOWNGRADING/DELIMITING INSTRUCTIONS

14. RELEASE AUTHORITY

Chief, Communications Division

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved for public release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE CENTRE, DIS NETWORK OFFICE, DEPT OF DEFENCE, CAMPBELL PARK OFFICES, CANBERRA ACT 2600

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CASUAL ANNOUNCEMENT Yes

18. DEFTEST DESCRIPTORS

CORBA (Computer architecture); Object-oriented systems architecture; Systems integration; Interoperability; Command, control, communications, computers and intelligence

19. ABSTRACT

Based on object technology, the OMG defines an Object Management Architecture (OMA) for the support of interoperable applications across heterogeneous computing platforms. The communication core of this underlying model is the Common Object Request Broker Architecture (CORBA) that provides a framework for flexible and transparent communication between distributed objects. The adoption of this approach eases software development by allowing interaction between reusable components through well-defined interfaces. In particular, applying CORBA technology to C4I problems in the military environment provides simple integration of legacy software and COTS software. This report provides an overview of the OMA, and describes in detail each component of CORBA.